2012

Semi-automated parallel programming in heterogeneous intelligent reconfigurable environments (SAPPHIRE)

Sean Stanek
Iowa State University

Semi-automated parallel programming in heterogeneous intelligent reconfigurable environments (SAPPHIRE)

by

Sean Stanek

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Program of Study Committee:
Carl Chang, Major Professor
Johnny Wong
Wallapak Tavanapong
Les Miller
Morris Chang

Iowa State University

Ames, Iowa

2012

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

PROJECT SUMMARY

Advancements in computer hardware technology continually provide faster and faster computational platforms. However, in recent years, as we come closer to approaching physical limits in making smaller (and faster) computer processors, focus has instead been turned toward including multiple processor cores in each device. While this technically allows more computational power in the same amount of time as compared with only one processor core, conventional software typically can only make use of a single processor. Multithreading is required for software to be able to effectively utilize multicore processors.

Goal: Our goal is to design and develop a middleware platform that supports stream programming and reduces the time and effort necessary to develop stream programs. The middleware determines a good configuration for each software component to exploit heterogeneity and parallelism of the hardware system. Software developers do not need to concern themselves with how and which computing device executes which component.

Contributions: We have three major contributions: (1) SAPPHIRE, a middleware for semi-automated program construction of stream programs based on data dependency matching. SAPPHIRE allows stream application development to be accomplished with significantly fewer lines of code and eases collaborative development. (2) A novel static task-scheduling framework for stream programs with heterogeneous implementation choices. We proved that the maximum load approximates makespan of a stream program to within a negligible amount of error. (3) EM-Capture, an automated real-time application using novel video analysis techniques for endoscopic video detection.

Impact: We contribute to three important areas of computer science: software design, biomedical image analysis, and high performance computing. In addition, our software enables automatic capture, analysis, and feedback of quality for endoscopic procedures that has never been possible before in practice. Our endoscopy software has analyzed over 50 billion frames and captured over 71,000 endoscopy videos in a real hospital setting. Our software has great potential to raise the quality of patient care through automated real-time feedback and documentation.

CHAPTER 1. INTRODUCTION

The computational requirements of newer computer programs have grown greatly over time; at the same time, the amount of computation a processor can do has also grown. However, the processing speed of individual processors has not grown as fast to accommodate new applications/programs desired to run on them. As a result, multiple processors are utilized for some of these programs. Unfortunately, special design and implementation must be taken into consideration for parallel computing.

Many early implementations of parallel computing involved working on identical computing nodes, both for the simplicity of constructing the parallel computing cluster hardware and for the simplicity of writing a parallel program. Since the end-purpose of parallel computing is to minimize completion time of a particular program, it is advantageous in program execution to utilize a wide variety of computing platforms (although perhaps more time consuming to design) such as GPUs (graphics processing units), FPGA (Field programmable gate arrays), custom-made processors, and clustered or networked computers. GPUs have become a major focus in recent years, constantly driven and improved by the demand for faster and faster 3D graphics in games, making their high-speed, massively parallel processing elements a good candidate for a multitude of general computational intensive problems. GPUs have also become commonplace in personal computers. Custom-made processors are also candidates for parallel computing (though not cost-effective), as they can work on similar problems as FPGAs, but take up less space and run considerably faster.

To efficiently utilize heterogeneous computing platforms, developers can utilize low-level vendor-specific APIs and libraries. For some applications, developers can use existing middleware packages such as MPI (Message Passing Interface) for common parallel programming data management and synchronization. With MPI, several copies of a program are loaded at the same time, one per processor. Each program runs in full parallel during its entire execution on independent data sets, with the exception of data dependency or synchronization. The MPI programming model follows a *data parallelism paradigm* (same processing task on independent data sets). It works well for problems in which data can be

split up and processed independently. However, only certain kinds of algorithms can easily take advantage of this paradigm.

Pipeline parallelism (*task parallelism*) paradigm involves the parallel execution of different tasks on different data sets. In an environment that processes continuously streaming data such as a stream of images, this is also called *stream programming*. This paradigm also has its limitations that programs must work on streaming data. That is, one component of the program processes a piece of data, passes its results on to the next component, which does its own processing and passes its results on to the next component, and so on. Instead of the first component remaining idle during the other components' executions, a new piece of data is fed to the first component. These components all constantly receive new data from their predecessors and all run in parallel with each other. FPGAs utilize this model of computation, performing lookups (computation) at every lookup table (LUT) on the FPGA all of the time at each clock cycle. CPUs themselves are an implementation of this paradigm, performing computation on the instructions and intermediate data at each stage in its pipeline. Important applications of stream programs are real-time quality monitoring of medical procedures, video surveillance for security, just to name a few.

Currently, it is time-consuming for developers to take advantage of heterogeneous computing platforms to efficiently run their stream programs. Our goal is to design and develop middleware named **SAPPHIRE: Semi-Automated Parallel Programming in Heterogeneous Intelligent Reconfigurable Environments** that supports stream programming and requires less time to develop stream programs. The platform does multithreading, allows dynamically loadable components of stream applications, and determines a good configuration for each software component to efficiently exploit heterogeneity of the hardware system. The software developers do not need to concern themselves with how and which computing device executes which component. This issue has not been investigated in the research literature. SAPPHIRE provides flexibility for developers of stream applications to get the most out of their computing platforms without having the developers to determine these configurations themselves.

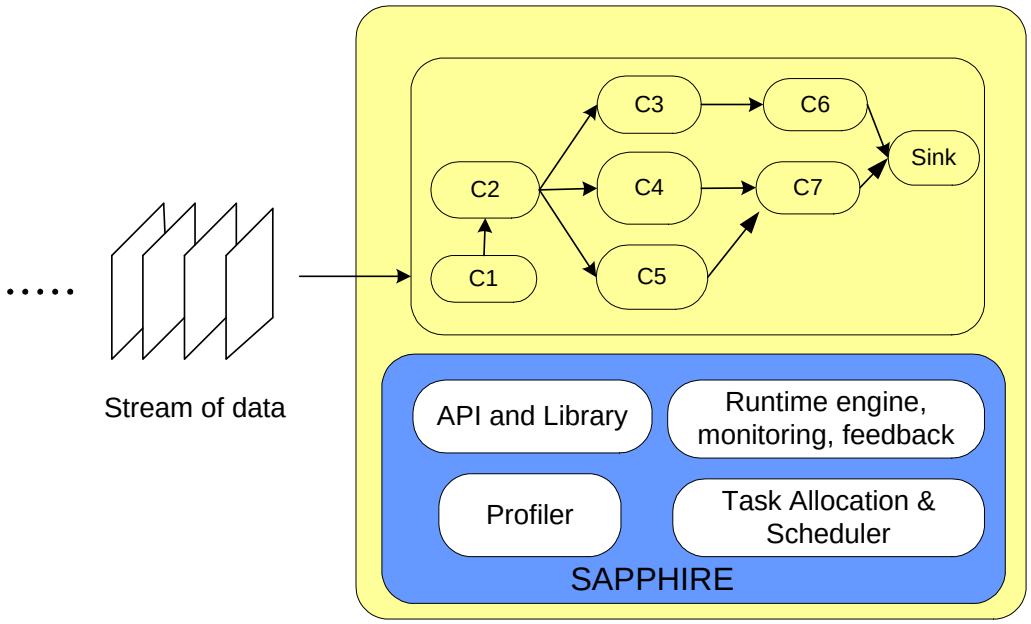

Figure 1.1: SAPPHIRE will support any stream program to utilize heterogeneous computing platforms. Ci are components representing different processing tasks. Ci forms a task graph that describes the data dependency among the tasks. For instance, C2 waits for the result from C1. C3, C4, and C5 cannot start until C2 finishes.

Our contributions:

1. SAPPHIRE – A middleware and software development kit (SDK) for semi-automatic program construction of tasks by data dependency matching. Unlike existing work, SAPPHIRE can support task-parallel stream applications that have precise requirements when accessing data. SAPPHIRE allows stream application development to be accomplished with significantly fewer lines of code and eases collaborative development.

2. A novel static task-scheduling framework for stream programs on a heterogeneous multiprocessor system. Our framework does not require unrolling an original PTG as in recent existing work, which would expand the problem size. Our framework supports heterogeneous implementation (HIT) choices. We formulate the task allocation problem with HIT support as a load balancing problem that optimizes the maximum load (execution time) among all the processors in the system. Given large

inputs, we proved that this maximum load approximates makespan of a stream program to within a negligible amount of error. This problem formulation enables us to have a simple heuristic load balancing algorithm called K-HIT that solves the formulated problem. For the final task scheduling, we propose a variant of earliest finished time first for stream programs called Stream-EFT.

3. EM-Capture – We developed novel algorithms and application for automatic endoscopy video analysis for endoscopic procedure detection. Our application does not require any human intervention, making it easy to use in a real hospital setting. EM-Capture is a novel automated endoscopy video capturing software. Our software has analyzed over 50 billion frames and captured over 71,000 endoscopy videos in a real hospital setting. An extension to EM-Capture, called EM-Automated-RT, provides real-time quality analysis and feedback for colonoscopy. EM-Automated-RT was developed as a collaborative effort using SAPPHIRE.

This dissertation contributes to three important areas of computer science: software design, biomedical image analysis, and high performance computing. In addition, our software enables automatic capture, analysis, and feedback of quality for endoscopic procedures in real hospital settings that has never been possible before in practice. Our software has great potential to raise the quality of patient care through automated real-time feedback and documentation. This dissertation resulted in several publications during the time of its writing [1][2][3].

The dissertation is organized as follows. Chapter 2 provides relevant background information and related work. In Chapter 3, we provide the overview of SAPPHIRE. In Chapter 4, we describe the design, algorithms, key internal data structures, and implementation in more details. In Chapter 5, we present our novel static stream task scheduling framework, involving theory, proofs, simulation, and a proof-of-concept implementation of a user-mode thread scheduler. In Chapter 6, we present our evaluation of SAPPHIRE using two real-world case studies, EM-Capture and EM-Automated-RT. In Chapter 7, we conclude the dissertation and describe future work.

CHAPTER 2. RELATED WORK

2.1 Program Construction

Although there are many ways to construct a program, there are some specific methods that we are interested in for our research. As a way to enable parallel computing, program construction methods that ease parallel program construction are of key interest. Using separable functions that can run independently of (and thus in parallel with) each other on different pieces of data creates an implicit opportunity for task-parallel computing. We will investigate various program construction methods that enable parallel in this way. There are some other unique advantages that these program construction techniques allow that we will also investigate.

2.1.1 DirectShow

DirectShow is a component of DirectX, a large multimedia library created by Microsoft for the Windows platform. The purpose of DirectShow is to enable multimedia drivers, devices, and software to work correctly with each other, regardless of the format of the data [4]. This is also sometimes referred to as device independence.

A software program creates and executes a DirectShow graph, either with a common programming language (many are supported) or as a visual graph. This graph consists of multiple filters (components), usually some input filter like a file reader, providing a source of data; some output filters, such as a video display window; and intermediate filters that can decode and convert the data to the desired output format, such as video codecs and color space converters. Filters are connected via input and output pins. Each pin can be queried for supported data formats that they accept. The programmer can manually set the data types of each pin and connect filters, or DirectShow itself can attempt to automatically connect filters, inserting converter filters when needed.

While made for multimedia and perhaps not necessarily intended as a programming language, it could be viewed as a kind of programming language, and there are some clear advantages to using a modular design like DirectShow. Device independence between modules, potential component optimization through task parallelism (the independence

between modules enables this), and graphical programming are some advantages. DirectShow has some disadvantages, particularly as a result of its intention to be used as a multimedia experience. For example, if some modules are too slow and start lagging behind, DirectShow may choose to skip processing some frames of data for certain modules. This may be disadvantageous for a scientific processing platform. Some further disadvantages are discussed in Chapter 3.

2.1.2 Circuit Languages

Some programs are constructed as a flow of data passing through functions as opposed functions working on sets of data. For constructing computer circuits, this is especially the case. On the electrical engineering level, every computing component is "running" all the time, as opposed a procedural program, which runs one line at a time. With a language like Verilog [5] or VHDL [6], we can construct a program where the data flows through simple functions in this fashion. We synchronize data inputs and outputs with the rise and/or fall of an externally generated clock signal. Not limited to silicon circuit construction, it is possible to create complex programs in the same fashion. Instead of being restricted elementary gate logic, we can utilize complex algorithms in their place.

The biggest advantage to constructing a program in this fashion is that all functions in the system are expected to run in parallel. A program written this way can implicitly take advantage of any multiprocessing available in a computing system, since all of its components can run in task-parallel.

2.1.3 Aspect Oriented Programming

While conventional programming relies on functions with their defined inputs and outputs, with functions running in the order in which a program is written, aspect oriented programming allows a programmer to statically or dynamically change the data routing of a function [7][8]. For example, a program might pass an image as an input to a file output function. Instead, the program might be changed to deblur the image before it is written out. Instead of explicitly calling a deblur function before calling the file output function, the programmer could instead attach a deblurring function to the input of the file output function.

The output of the deblurring function would be the new input to the file output function. Any time any part of the program calls the file output function with an image, the image is first deblurred before it is actually passed to the file output function. Although aspect-oriented programming is actually much more than the simple example presented, this is the main concept of interest to us.

2.2 Parallel Computing

In this section, we discuss common types of parallel computing paradigms and parallel computing middlewares.

2.2.1 Data Parallelism

Message Passing Interface (MPI)

MPI [9][10] was introduced in 1993 was one of the earliest standards developed toward making development on distributed computing and distributed memory systems easier. It is implemented as an API in the programmer's language of choice (typically C or Fortran). Most of the API functions are simple: delegating and identifying computation nodes; sending and receiving data from one node to another, or perhaps any number of nodes to any other number of nodes; and providing barrier synchronization. Although task parallelism would be possible with MPI, its design leans toward making data parallelism easier. The data distribution and collection functions are especially useful for distributing data equally among all nodes.

CUDA

NVIDIA's CUDA (Compute Unified Device Architecture) is a parallel programming infrastructure especially designed for their graphics processors [11]. Programmers write a GPU "kernel" (a small data-parallel program) in a modified C language that provides some GPU primitives. The CUDA compiler compiles the kernel code to the native GPU's machine code and the non-kernel code to the host CPU's machine code. When a program is executed, the CPU code tells CUDA what GPU code to load, and then executes it on the massively parallel GPU computing cores. These are run in single-instruction multiple-data (SIMD)

fashion in “warps” containing 32 threads which each execute the same instruction on 32 different pieces of data. A collection of warps are run inside a “block” (up to 768 threads). A collection of blocks runs inside a “grid” that executes the same code on a “multiprocessor.” An individual multiprocessor can schedule different kernels to be run, but only one kernel is actually run at any given time for a multiprocessor (similar to how an OS does task/context switches). Separate multiprocessors inside a single GPU device can run different kernels simultaneously. In recent years, a significant effort in both academia and industry has been made to utilize GPUs for general purpose computing in several application areas such as image analysis [12][13][14][15], computational biology and chemistry, and simulation. To fully utilize the power of GPUs, developers have to take into account several aspects of the underlying device architecture as well as program characteristics in their design. These aspects include data dependency, efficient memory access (e.g., use of fast shared memory, memory coalescing and banking), task allocation, and task scheduling to reduce the required bandwidth between CPU RAM and GPU memory. An analytical model for a GPU architecture has also been proposed [16].

2.2.2 Task Parallelism and Stream Programming

Microsoft DirectShow, previously mentioned as a method of program construction, has the side effect of implicit task parallelism due to its programming model. Because each DirectShow filter is a computationally independent and separated component from other filters, each filter automatically has the capacity to run in task-parallel. The programmer that uses DirectShow does not necessarily know how DirectShow schedules the execution of each filter. Some filters might run in serial and some in parallel, and some might even have their data dropped if DirectShow feels that there is insufficient CPU time to run all filters in the time allotted. This real-time ability of DirectShow is also an important aspect of task-parallel scheduling of computationally intensive programs.

2.2.3 Hybrid Task and Data Parallelism

StreamIT

MIT's SteamIt introduced in 2002 is a programming language constructed for the sole purpose of describing streaming programs [17]. It works similarly to a circuit language, describing components, the inputs and outputs they read or write, and the how the components connect to each other. The StreamIt compiler reads source code, determines a stream mapping from that code, and can potentially find the most optimal, finest grain (down to the arithmetic level) parallelism possible. Because it is its own language, this means that a programmer must learn a new language and convert most of his or her code to the StreamIt language. Although it would be possible to link in external code in the linking stage of code generation, only code written in the StreamIt language benefits from the fine-grained streaming parallelism optimization.

OpenCL

Khronos Compute Working Group introduced OpenCL (Open Computing Language) in 2008. OpenCL [18][19] is a recent specification and framework for developing heterogeneous programs with as much cross-platform support as can be expected in a heterogeneous environment. It provides functions for setting up program components, queuing program communication flow between components, and executing the program as a whole. It provides some intrinsic data types for commonly misrepresented data types between platforms (e.g., integer size). The goal of OpenCL is to allow developers to write a heterogeneous program in data parallel, task parallel, or any combination of either.

Other

Chen et al. proposed a system for allocating tasks on GPUs at a finer granularity than that normally allowed by CUDA or OpenCL by using a job queueing system [20]. Jobs are submitted by the host system to a job queue in memory, where they are run in granularities of thread blocks by their proposed scheduler. In order to execute properly, all task kernels must be combined along with the job scheduler and loaded to the GPU at the same time, where the custom GPU thread block scheduler dispatches tasks from the job queue.

2.3 Multiprocessor Task Scheduling

Multiprocessor task scheduling has long been studied. Dutot et. al. provides a summary of the related work in this area [21] as follows. (1) Makespan – the application execution time is often used as the performance metric. (2) The task scheduling problem on homogeneous platforms is NP-Complete. (3) Algorithms with performance guarantees where the performance metric was defined as the maximum ratio between the produced makespan and the optimal makespan has been developed. Different heuristic methods such as variants of list-based scheduling and simulated-annealing based scheduling were proposed [21]. List scheduling-based methods are most popular.

The majority of previous work focuses on homogeneous platforms or heterogeneous clusters of homogeneous systems within a cluster. Furthermore, platform heterogeneity is in terms of variety in CPU processing capabilities or communication among various sites. The majority of previous work does not focus on task scheduling of stream programs.

2.3.1 Homogeneous Task Scheduling

Given (1) some program tasks, each with some inputs and outputs, where some task's inputs (dependencies) may be derived from some previous tasks' outputs, (2) a set of identical processors with which to run the tasks on, and (3) their expected execution times on the given processors – we wish to find both (1) the optimal allocation of a number of processors to tasks and (2) the optimal placement which set of processors will run each task, such that the result minimizes the overall execution time. Some task scheduling algorithms will do the allocation and placement in separate stages (usually considered easier since the allocation can be done heuristically without finding the placement ahead of time), and some algorithms will attempt to do both simultaneously. Because multiprocessor task scheduling is NP-Complete [22], task scheduling algorithms are usually heuristic algorithms in practice.

Scheduling Algorithms

The FIFO (first in first out) scheduling algorithm [23] is the simplest of scheduling algorithms. As soon as a task's dependencies are satisfied, the task is queued to run on the next available processing node(s). When a task finishes, its outputs are used to satisfy the

dependencies (inputs) of other tasks. When a new task’s dependencies are all satisfied, the task is queued. Whenever a processor is free, it may be used to schedule the next task in the FIFO queue, and the process repeats. The problem with this algorithm, while extremely simple, is that it is greedy and may perform poorly or allow a slow task to hog processor resources when a different task may have been better overall (in terms of minimizing overall program execution time). This is an example of an algorithm that performs allocation and placement at the same time.

Round robin (RR) [23] is a naïve algorithm that creates an absolute ordering of tasks (not necessarily in an order based on any kind of optimality) and then schedules each of them one at a time until all have been scheduled. In a stream program, the amount of time given per task could be one iteration's worth of work. Once all tasks have been given their first block of time, the tasks are once again scheduled in the same order, and the process repeats.

Earliest finish time (EFT) [23] first creates an ordering based on the earliest possible finish time of each task, which changes based on when a task's dependencies can be satisfied.

Dynamic list scheduling (DLS) [24] is a more complex algorithm than RR or EFT in making use of more global information. A task's static level is the critical path length from a task's node to the sink node (without regard for scheduling, only satisfying dependencies). The static level of a task minus that task's earliest start time yields a dynamic list ordering by which tasks are scheduled. This approach favors tasks with a higher static level (we need to complete these tasks in order to reduce global execution time) and an earlier start time (in order to schedule tasks soon after they become available).

2.3.2 Heterogeneous Task Scheduling

In heterogeneous task scheduling, we remove the constraint that the processors be identical. Homogeneous processors make scheduling easier, since it makes little difference as to which processors you allocate to a task: they will all take the same amount of time to work on a task. In heterogeneous task scheduling, different processors may take a different amount of time to finish the same task. Because of this possibility, it does matter which processors you allocate to a task. For example, allocating a long task to a slow processor and a short task to a fast processor may not be as effective (in terms of wall clock time) as allocating the long

task to the fast processor and the short task to the slow processor. In the later case, the end time of the longest task would have been earlier (and thus, usually preferred).

One such example is that of [25]. A two-stage scheduling algorithm is utilized: first allocation of a number of processors to each task, followed by the placement of each task onto a set of that many processors. To account for heterogeneity, an attempt is made to first convert the problem into a homogeneous task-scheduling problem. Suppose we have a set of N processors that run at different speeds. Instead of attempting to schedule on N heterogeneous processors, a uniform reference-speed virtual processor is established. The relative computing power of each heterogeneous processor is converted to the speed of the uniform processor. For example, a 1.5GHz processor may be equal to 1.5 1GHz virtual processors. Upon summing the number of virtual processors, where M is the sum of virtual processors, we now use a homogeneous scheduling algorithm on those M processors.

In the allocation step, a processor is allocated iteratively to the critical path of a given task graph. The critical path of a task graph is a path from source to sink with length equal to the makespan. Reducing the overall execution time of the program means reducing the makespan, and this cannot be accomplished without reducing the execution time of each critical path. Thus, to determine how many processors to allocate to which tasks, a critical path is identified, and a task along that critical path is allocated more computing resources. This is done until all processors have been allocated.

In the placement step, we already know how many processors we will use for each task. So, we can estimate the duration each task for that number of processors allocated to it. The task that has the earliest finish time and is also ready to execute (all dependent tasks have finished and the number of processors that the task has been allocated are available) is given its number of allocated processors. As tasks are completed, dependencies are satisfied, and more tasks may become ready to execute. This is repeated until all tasks have been scheduled. In the end, the virtual processors are translated back to real processors to define the final mapping of processors to tasks.

Many varations on scheduling algorithms have been investigated for both homogeneous and heterogeneous scheduling [26][27][28][29][30], specifically including list

scheduling [31], dominant sequence clustering [33], genetic algorithm based [34], dynamic level scheduling [35], best imaginary level [36], mapping heuristic [37], levelized min time [37], scalable task duplication [38], fast critical path [39], fast load balancing [40], and heterogeneous earliest finish time first [41]. A summary of algorithms is provided in Table 2.1.

2.3.3 Stream Task Scheduling

The aforementioned algorithms can all be extended to stream program scheduling by a method called *graph unrolling*: unrolling a stream program's parallel task graph by potentially millions of iterations, so that each iteration of each task has its own node in the task graph [24]. Then, a chosen task scheduling algorithm is used on the larger, unrolled graph. Although this is a working general solution for stream task scheduling, it is inefficient as it requires expanding an already NP-Complete problem by many orders of magnitude.

Cyclic scheduling

Cyclic scheduling is a well-studied field in manufacturing that is also applicable to stream task scheduling. There are many variations of cyclic scheduling [42], but they share the same objective of minimizing the cycle time – the amount of time between the start and end times for each processor of a static task schedule – after which, the cyclic schedule can be repeated. See Figure 2.1(d). The cycle time is different than the makespan in that the makespan is the time difference between the earliest start time and latest end time of all tasks across all processors, whereas the cycle time is computed as the largest difference across all processors between start and end times within a single processor. This may yield a smaller and more optimal schedule for repetition as the cycle time may often be less than the makespan.

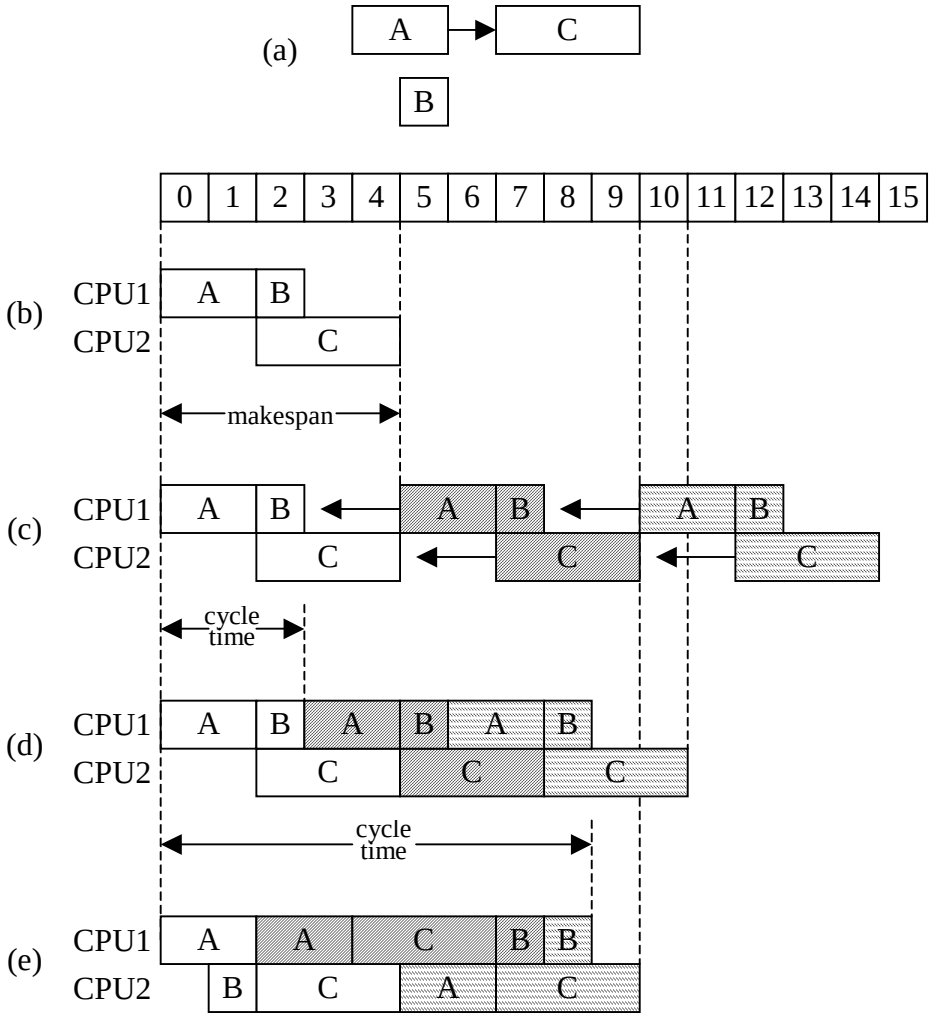

Figure 2.1: Example of various scheduling algorithms. In this example, there are three tasks, A, B, and C; with execution time costs of 2, 1, and 3, respectively. The only dependency is that task C depends on task A (a). There are no other restrictions (e.g., CPU bindings, self-dependencies). In (b), a single iteration of this task graph is scheduled. Because task C cannot start until task A completes, task C cannot be scheduled until time 2. Task B could be scheduled at any of the unused times, and its location in (b) is arbitrary. The total execution time (makespan) for this single iteration is 5. In (c) through (e), we want to schedule the task to run three times. The iteration number of each task is indicated by the solid diagonal lines for iteration 2 and dashed diagonal lines for iteration 3. A naïve approach is performed in (c), where we run schedule (b) to completion, then rerun it to completion, and finally repeat it a third time. The total execution time for (c) is 15. The arrows in (c) indicate potentials for optimization, as this is mainly unused execution time. In (d) we show the cyclic schedule for these three iterations. As there are 2 time units free on CPU1 after task B completes, and 2 time units free on CPU2 before task C starts, the next iteration can fit if we slide it 2 time units earlier. This can be repeated for as many iterations as necessary. The cyclic schedule, specifically, is just a shorter schedule (e.g., a single iteration, such as (b)) that will be repeated every 'cycle time' time units. In (d), the cycle time is 3 time units, as the short schedule can be repeated every 3 time units. For scheduling 3 total iterations, the makespan is equal to 11 time units. In (e), we have unrolled the graph and scheduled all tasks together, while still allowing for the dependency that each iteration's task C be schedule after it's task A. This expands the problem size, but yields a more optimal solution of 10 time units. Although not shown here, it would be easy to see that (e) could be used as a schedule for cyclic scheduling, where 3 iterations could be performed with a 3-iteration cycle time of 8 time units. This would reduce the average time per iteration down to 8/3 or 2.67 time units.

The variation chosen may alter the optimization goal or change the set of formal restrictions (e.g., whether each iteration is independent of each other). Given a task-to-processor allocation ahead of time, an optimal cyclic schedule can usually be determined in polynomial time [42]. Otherwise, this usually becomes an NP-Complete problem (depending on a set of assumptions and constraints). Although a schedule may be optimal given a set of formal restrictions, the schedule might not be optimal with a different set of restrictions. A major advantage of this approach is that only a small static schedule is generated for one or a few iterations, reducing the problem size dramatically from the full graph unrolling method.

2.3.4 Heterogeneous Stream Task Scheduling

Satish et al

The most related work to our research is that of Satish [22]. He investigated a heterogeneous task-scheduling problem on multicore CPUs and a GPU for stream programs. His method aims to partially unroll only a few (one to ten) iterations of the original PTG rather than fully unroll the PTG across potentially millions of iterations. This restricts the problem size to the number of tasks times only a few iterations, rather than having a million times blow-up in problem size as in full PTG unrolling. A scheduling algorithm (such as DLS) is then performed on this partially unrolled graph to achieve a schedule of only a few iterations. Figure 2.1(e) might be the result of a schedule with the original parallel task graph unrolled three times. The schedule is then concatenated to itself, repeated end-to-end, until all iterations of a fully unrolled task graph are accounted for. While being less optimal since the entire problem set is not used, the resource requirement to compute a schedule on a partially unrolled graph is significantly less.

He applied a variety of heuristic methods including dynamic list scheduling (DLS), simulated annealing (SA), and a decomposition-based constraint programming approach (DA) on the partially unrolled graph. In many cases of his results for some algorithms, the true optimal mapping was not found, and usually considerable computing time was spent just

to achieve the results they did compute. The time required to make use of the model might be much more than desired for the heuristic solutions described.

2.3.5 Windows Task Scheduler

In standard operating system multitasking, fairness is usually preferred in design more than overall completion time for a set of tasks. This is done in practice for many reasons. An operating system usually does not have a priori knowledge about task durations, and so, cannot predict how long a task will take. Also, task dependencies are usually not explicitly specified to the OS. When user interaction exists, the OS usually does not know which task a user prefers to finish first (although Windows handles this somewhat, as discussed later). Response time may also be an issue, where a cooperative task scheduler could have a single long-running thread blocking many other threads from completing. So, overall completion time, which may be impossible without a priori knowledge and significant metainformation about the tasks to be run, is usually sacrificed for fairness.

In Windows, preemptive multitasking is used at the granularity of threads (not processes) [43]. Each thread is given a certain amount of time to run, called a time slice. After a thread has run for its full time slice on a given processor (if it has not already yielded cooperatively), it is forcefully preempted by the operating system and a different task is swapped in to run on that processor. On a modern Windows system, the time slice is approximately 15 ms. Round robin scheduling is typically used to provide overall fairness. Foreground threads are given three times the normal time slice duration in order to prefer applications the user is actively using to increase perceived system responsiveness. Windows also supports process and thread priorities, which together form an overall thread priority level, which is used to allow some threads to run before others. A "real time" priority is allowed, which prevents other threads from preempting a time critical thread. Windows also employs many other complex scheduling heuristics that may temporarily boost a thread's priority to reduce resource locks that could cause performance problems (for example, a mutual exclusion lock immediately followed by a task switch could cause other threads that also depend on that mutual exclusion object to waste time, as the lock has already been acquired by a previous thread).

2.4 Endoscopy Video Analysis

Colorectal cancer is currently the second leading cause of cancer related deaths in the United States, just behind lung cancer. It is estimated that more than 141,000 people in the US were diagnosed with colorectal cancer in 2011, and over 49,000 people died from it [44]. The standard procedure for identification of colorectal cancer and removal of precancerous lesions is a colonoscopy [45]. In this procedure, an endoscope, which is a long tube with a tiny video camera and wide-angle lens on the tip of it, is inserted into the rectum and advanced through the colon to the cecum or terminal ileum. Then, the endoscope is slowly removed from the colon by retracting it, while the endoscopist carefully inspects the inner lining or mucosa of the colon. Abnormalities such as polyps that may develop into cancer can be removed during the examination. In some cases visual documentation of findings or the applied therapy is desirable; recording of images or video of the findings or therapy is then performed allowing repeat inspection at a later time.

Over 14 million colonoscopic procedures are performed annually [46]. The current Medicare guidelines suggest that each US citizen undergo colonoscopy at least once every 10 years starting at age 50. While colonoscopy has contributed to a decline in the number of colorectal cancer-related deaths, recent data suggests that there is still a significant miss-rate for the detection of even large polyps and cancers [47][48][49]; the colonoscopy adenoma miss rate may be as high as 4% to 12% [50]. Evidence suggests that endoscopist-related factors influence polyp detection rate. For instance, a landmark study published in 2006 reported that polyp detection rate of screening colonoscopy increases with increasing time spent during the withdrawal phase of the procedure [51]. Other factors may also influence polyp detection rate, such as speed of withdrawal, effort to visualize all of the mucosal surface, bowel preparation and experience of the endoscopist.

2.4.1 Picture Archiving and Communication Systems (PACS)

The most related work is in the area of video capture and PACS (Picture Archiving and Communication Systems) for endoscopy. Typically, the endoscope video signal is directed into a video capture device on a computer, where the video from the endoscope can be saved as individual snapshot images or captured as video. Video and images are typically

captured manually, either through interaction with the computer, by using a button on the endoscope, or by using a special foot switch. Many of these video capture systems are connected to electronic medical record systems (e.g., Cerner, Pentax, Gtech Information Systems). There are no practical automated tools that allow one to keep very precise records reflecting a colonoscopy exam.

2.4.2 Scene Segmentation and Object Detection for Colonoscopy

Endoscopic video detection can be seen as a specialized scene segmentation algorithm that segments a sequence of frames into two types of scenes: procedure scenes and non-procedure scenes. There are many scene segmentation techniques that have been proposed for specific application domains such as news and movies [52]. Scene segmentation has never specifically been applied to endoscopic video detection.

In the Wireless Capsule Endoscopy (WCE) field, some existing work focused on dividing a procedure video into several segments corresponding to major anatomical landmarks [53][54][55] or groups of similar frames [56]. These techniques assume that the input video is a real procedure video.

Other previous work on endoscopy image analysis are for polyp detection [57][58][59][60][61][62], automated objective quality measurements of colonoscopy based on motion features, quality of images, and types of clear images [63], presence of a clearly seen appendiceal orifice [64], and 3D reconstruction of the colon structure [65] and colon surface [66][67][68].

Manual analyses for specific features in recorded videos would require an experienced endoscopist or assistant to review every procedure; this would be redundant, expensive, difficult to implement, and subjective. Indeed, to conduct quality control tests for every procedure, automated analyses would need to be available.

Table 2.1: Overview of scheduling algorithms

Platform	***For non-stream programs***	***For stream programs***
Homogeneous	*First-in first-out (FIFO)* *Earliest finish time (EFT)* *Round robin (RR)* *Shortest job first (SJF)* *List scheduling*	*Loop unrolling: slow* *Partial loop unrolling: fast* *Cyclic scheduling: slow, but yields small solution*
Heterogeneous	*Dynamic level scheduling (DLS)* *Critical path on a processor (CPOP)* *Generalized dynamic level (GDL)* *Best imaginary level (BIL)* *Task duplication scheduling (TDS)* *Mapping heuristic (MH)* *Levelized minimum time (LMT)* *Heterogeneous earliest finish time (HEFT)* *Fast critical path (FCP)* *Fast load balancing (FLB)* *Heterogeneous N-predecessor decisive path (HNDP)*	*Loop unrolling: very slow* *Partial loop unrolling: fast, good results, does not support heterogeneous choice well* *Cyclic scheduling: slow, but yields small solution* *Stream-EFT + K-HIT: very fast, good results, no unrolling necessary for NP-Complete part, supports heterogeneous choice*

CHAPTER 3. OVERVIEW OF SAPPHIRE

The primary goals of SAPPHIRE are to (1) provide an implicit multithreaded environment for complex data analysis, especially for stream programs such as video processing, (2) ease the programming process for writing such a program, (3) ease the collaboration process between parts of a large program, (4) modularity, (5) program scalability, (6) extensibility and configurability, and (7) provide a framework for a proof of concept for our research.

We designed our middleware from scratch, using years of past experience developing a multiprocessing application, as well as coming up with solutions to numerous issues that those previous generations of software encountered. The design methodology and implementation address these goals. We support any environment or extension capable of generating or being called from executable code, including popular APIs such as CUDA and OpenCV [69]. Thus, we can generate heterogeneous programs to take advantage of GPU computation. We support both task-parallel and data-parallel design methods, specifically for stream programs. We attempt to ease the process of porting existing code by allowing the programmer to choose the language they want (as long as it is capable of generating executable code – although additional header files for their language may be needed). We attempt to simplify most of the redundant and complex requirements of related work, while also correcting some serious problems, such as DirectShow's synchronization problems. These problems are discussed in more detail in the next section.

3.1 Design and Overview of SAPPHIRE

Several aspects of program construction were considered when creating SAPPHIRE. In particular, we wanted to remove as much redundancy of the development process and automate as much as possible, particularly in regards to multithreading and collaboration. The result is a very modular system consisting of well-defined tasks and data.

We have designed our middleware to take advantage of both task and data parallelism. The programmer creates a program based on several separable tasks, which we

call modules. These modules may receive any number of inputs and generate any number of outputs (as memory allows). Inputs and outputs are formatted pieces of data, which we call packets. A whole program is constructed by putting together invididual modules that, when combined, represent the overall intention of the program. Some initial inputs and configurations are given to the modules, which provide a flow of data through the program, eventually resulting in some final outputs representing the result of the program.

Modules do not need to worry about how or where they receive their inputs from – they only need to request types of data. Then, the modules will automatically receive those types of data packet-by-packet through a callback function. Some additional synchronization setup functions are provided to synchronize different types and multiple packets worth of data. This design frees up the programmer from complex tasks involving communication and manual synchronization.

For example, an MPEG video compressor may consist of several components: (1) converting a source video to the YUV colorspace, (2) estimating the motion of 2D blocks of image data between frames, (3) transforming blocks of image signal data with the discrete cosine transform (DCT), (4) quantizing the resulting DCT to provide input to (5) entropy encoding that compresses individual blocks of video data, and (6) combining all the outputs into a final MPEG video stream, which may be written to a file. The above components work in a streaming manner, repeating with each frame of video provided as input. In this example, there are divisible components that can perform tasks independently of other tasks, as long as they have different pieces of data to process. The different pieces of data are the different frames of video as well as the intermediate results of each component. The quantizer outputs the quantized DCT matrices as some form of data which is then used as input by the entropy encoder.

Every piece of data that is used as input or written as output by a module is explicitly specified as a *data type*. For example, VIDEO could be a data type representing the video stream. QUANTIZED_DCT could be a data type representing the output of the quantization step. These pieces of data are put into a *packet*, which contains the data, a description of the data (called the metadata), and some bookkeeping information about the packet itself. The

metadata defines a structure for the data of a packet that is used by the modules. For a VIDEO packet, this includes the width, height, stride, etc., as well as a pointer to the video data itself. A module calls SAPPHIRE to create the packet, but defines its structure and fills it out with data itself. The packet is then pushed to SAPPHIRE, where all the routing of data is done automatically by the system. The exact format of a packet's metadata is specified by a module programmer for each type of data. A structure for this format is provided by the module programmer to other module programmers that need to use that data type, typically as a header file. Some documentation should also be written by the programmer so that exact usage details are provided in order to avoid unexpected misuse of data.

By design, no module generally needs to explicitly communicate with any other modules. Because the middleware handles nearly all of the communication aspect, modules work more independently and thus are able to take more advantage of parallelism. Removing the need for explicit synchronization and communication between modules as well as implicitly giving each module its own thread makes creating a multithreaded application much easier from a developer standpoint.

The middleware provides an API for modules to communicate with the middleware (as opposed to modules communicating directly with other modules). An interface is also defined for modules – each module implements several callback functions, which the middleware calls in order to do things like initialization and commucation. The *Register* function, implemented by each module, is called by the middleware to give each module the chance to request types of input data it can process and specify what data types it will output. After all modules' Register functions have been called, the middleware knows every input and output data type that each module needs. From this, the middleware can build a complete picture of all the communication and routing involved between modules for packets. This can be represented by the parallel task graph.

Table 3.1: List of methods to be implemented by module developers

Methods implemented by each module	***Purpose***
emmRegister	*Receives configuration data from the middleware*
emmStart	*Middleware notifies module to start execution*
EmmData	*Middleware submits single packet of data to module*
EmmStop	*Module should stop execution*
emmShutdown	*Module should free all resources*

Table 3.2: List of core middleware functions by type (a more detailed list of API functions can be found in Appendix A)

API functions provided by SAPPHIRE	***Purpose***
Registration:	
emcAddInput	*Request data type as an input*
emcAddOutput	*Register data type as an output*
emcAddMuxInput	*Request logical grouping of data types as input*
Communication:	
emcNewPacket	*Create a new data packet*
emcPushPacket	*Submit a packet of data to the middleware*
emcReleasePackets	*Release a reference to one or more data packets*
emcReleaseMuxPackets	*Release a set of data packets grouped by mux*
Synchronization:	
emcMuxPacket	*Insert a single packet into a mux and check for mux satisfaction*
emcInheritPacket	*Inherit synchronization properties from a parent packet*
emcSetFinished	*Set a source module to a finished state*

The core of execution of each module is the *work loop* – a loop that continuously waits for more data to become available, and then processes it. The work loop along with all its synchronization is implemented within SAPPHIRE itself (not each module individually), so the programmer has less to worry about. The *Data* function is implemented within a module by the programmer, where it is called by the middleware when data becomes available. After a packet has been pushed to the middleware by a module, the middleware routes that packet to every module that requests it through the Data callback function. Once

pushed, this packet is read-only, so that every module can then process it in parallel of other modules. A module's result of the processing is in the form of a new packet, which is pushed back into the system via a call to the middleware's *PushPacket* function. Once all pieces of data have been processed, the middleware finishes up and notifies each module to shut down before shutting down itself.

Repetitive obligatory code that might normally appear in a multithreaded environment where synchronization is important is minimized by design. The middleware handles most of the synchronization. Also, for other common repetitive tasks, the middleware either handles or otherwise provides easy interfaces to make programming less tedious. A bare-bones SAPPHIRE module could be written in about 10 lines of code, although this would not accomplish much without actually processing data received. As the incoming data packets are immediately exposed by the design of the middleware, it also takes relatively few lines of code to be able to process the data and write new data back to the system.

In order to create an actual program from a set of modules, a configuration file is specified which consists of a list of of modules and their parameters, inputs, and outputs. SAPPHIRE loads this configuration file, which in turn loads each module, passing along the parameters from the configuration file. The modules perform individual tasks in a "black box" methodology, while the configuration file specifies which components to use and how those components make up the larger program. Overall, the design of the system actualizes implicit parallel processing, speeding up the programming process, and making collaboration easy.

We improve upon related work, such as DirectShow, by vastly simplifying the development process. We remove the need for overly complex APIs and COM (Component Object Model) as well as removing the complex synchronization problems and negotiation of data types. A simple module with our middleware can be written in about 15-30 lines of code, whereas a simple DirectShow filter like the SampleGrabber filter takes several hundred lines of code. We also provide more guarantees in synchronization and completion of tasks, where DirectShow may never correctly stop at the same frame being processed for every

task, and is overly aggressive toward skipping frames when processing appears to be lagging behind. This is mainly because DirectShow is intended for use as a multimedia experience (where a loss of a frame every so often is not noticed by the user, such as watching a movie); it is not necessarily designed for use as a precise scientific analysis framework. A brief comparison of SAPPHIRE and related work is summarized in Table 3.3.

Table 3.3: Comparison of features among multiple toolkits that support data or task parallelism

Criterion	***MPI***	***CUDA***	***StreamIT***	***DirectShow***	***SAPPHIRE***
Programming Method	*API+Library*	*Extended-C + Compiler*	*StreamIt Lang. + Compiler*	*API+Library*	*API+Library*
Types of Parallelism	*Data*	*Data*	*Task+Data*	*Task*	*Task+Data*
Program Construction	*Explicit*	*Explicit*	*Explicit*	*Semi-automatic*	*Semi-automatic*
Accuracy in Data Processing	*Accurate*	*Accuracy often sacrificed for speed*	*Accurate*	*Skip some frames*	*Accurate*
Memory Leak Assistance	*No*	*No*	*No*	*No*	*Yes*
Performance Profiling	*3rd Party*	*Yes*	*Unknown*	*No*	*Yes*
Debugging Assistance	*Debugger*	*Debugger*	*Unknown*	*Debugger*	*Some built-in assistance; Other debuggers supported*

3.2 Features and Strategies

Unlike traditional multiprocessing middlewares, SAPPHIRE requires very little effort from the programmer in handling communication and synchronization. In programs that do their own multiprocessing, typically the programmer handles some complex synchronization issues, which in turn allows safe communication between modules. The code that handles these issues and communication may need to be replicated with each additional thread or task added to the system. For large programs, this is very repetitive and may require significant

work. However, for programs that utilize our middleware, this is unnecessary, as SAPPHIRE handles this internally. SAPPHIRE is also additionally designed to be able to handle the stream programming paradigm and implicitly take advantage of the added parallelism it provides.

SAPPHIRE is able to automatically build a program simply by listing a set of modules, parameters, and desired initial inputs and final outputs. Furthermore, the listed modules then provide their desired inputs and outputs. This information is enough for SAPPHIRE to construct a parallel task graph which can be used to determine all the synchronization and communication necessary for program execution. This collective information, as well as some additional API functions within the SDK, provide a powerful development and runtime environment for multiprocessing programs without a lot of effort on the part of the programmer.

3.2.1 Semi-Automated Program Construction

We further explore the stream programming paradigm and show some specific advantages that it allows. We make some further refinements to the realization of this paradigm by showing how to construct a new program from predefined components without the need for additional code. Program components (tasks) are automatically connected together based on their inputs and outputs, and run in parallel. Our method also allows for the pruning of unneeded code and data from the program.

Program Graph Construction

In constructing a program that conforms to the stream programming model, it may be useful to view a program as a directed graph since this representation provides a useful way to model aspects of a program. Individual, separable components or tasks can be viewed as nodes of a graph, and communication between components can be viewed as edges. Nodes can be labeled with costs associated with the execution time, and edges can be labeled with costs associated with communication time.

For a streaming program, working on n data sets in its data stream may be represented by a single node per task, where each node processes up to n data sets each over the course of

a program. Or, the same program could be represented by an “unrolled” version of the graph, consisting of up to n of each task, where each node processes only a single data set. This representation can become exceedingly complex when large amounts of data are used.

Data Dependencies

In the stream model, we recognize that every component in a program performs the same repetitive task on different pieces of data, much like an assembly line. Each task expects certain types of input data and produces specific types of output data. The input can come from program arguments, interactive user input, or the output of another task. Most parallel computing middlewares provide a method to the application programmer to pass the data from one task to another. This would typically be done manually by writing code to call the middleware API, sending and receiving or scattering and gathering data to and from other nodes. Depending on the application, the sending and receiving of data can become increasingly complex with the number of tasks in the system.

Instead of manually coding the passing of information to explicit nodes, suppose that instead, each task simply registers with the middleware what types of data it will accept and what types of data it will produce. A computer algorithm can very easily match up the outputs of any task to the inputs of any other task. This frees the application programmer from having to do explicit communication between tasks. Instead, the application programmer would do whatever processing needs to be done in a given task, and then simply write the data out to the middleware, along with the data type. The middleware searches for tasks that would like to receive that type of data and sends the associated data to those tasks. This is precisely what SAPPHIRE does.

Compared to conventional message and data passing in parallel computing middleware, this is very much like a broadcast operation. However, the targets of the broadcast are more limited and implicit to the application, making the software design simpler. Additionally, the middleware has the opportunity to automatically optimize the communication of the entire system. For example, using shared memory when possible to avoid the communication cost almost entirely, or perhaps it can choose good spatial locality for faster data transfers. Designing the middleware in this way also allows the application

programmer to design the software with more modularity than calling explicit communication functions. A side effect of modularity is that the separable tasks are much more easily run within a stream computing system – the stream model prefers modular tasks. As long as two tasks are separable, they can be run in parallel in some fashion, even if it is within a different iteration of processing.

Programming Methodology

We consider modularity to be important to the programming paradigm. Although programs are often designed with modularity in mind anyway, modularity is also a way to enforce separable tasks. By designing each task modularly, developers implicitly improve on the amount of parallelism that can be achieved.

An interesting caveat of having a modular program with automatic data dependency matching is that a full program can be constructed by specifying desired final outputs, possible module configurations, and program arguments or inputs. The framework can construct an actual program based on a library of modules available to it and the desired parameters. The modules capable of writing the final outputs are added to the program, modules with specific configurations are added and configured, modules related to the program arguments and inputs are added, and then the framework can automatically find, add, and configure all the modules in between (using some defaults or automated configuration), to create a complete program. Little programming knowledge would actually be required to construct a program in this fashion; only a few settings and final desired output need be entered into a configuration file.

3.2.2 Module Implementation

Implementing a module for the programmer is similar to writing a class in a programming language. Each module is implemented as a dynamic-link library (.dll) or a shared object (.so) file, which may be written in any programming language capable of producing such a file. A module implements and makes available five functions for the middleware to call: Register, Start, Data, Stop, and Shutdown. The Register function is called by the middleware to query for information such as a module's inputs, outputs, and version

information. The registration information describes the functionality and requirements for each module. Each module is then assigned its own thread. The Start function is called by the middleware to tell a module it may start processing. Additional threads can be created for additional parallelism if desired. The Stop function is used to tell a module it should stop producing new source data, but that it can finish processing its remaining data. The Shutdown function is called as a final cleanup and destruction of the module.

Actual processing is handled by the Data function as a callback. As a packet of data that the module is listening for becomes available, it is automatically passed to the Data function on that module's execution thread. The module may then decide to process it, buffer it and wait for more data before processing it, or release it. Some tasks may require several pieces of data before processing (such as computing the motion between two video frames). In other situations, the programmer might want to wait for several different data types to be available before processing any one by itself (such as an MPEG encoder writing both video and its corresponding audio to a file). The middleware's API handles these kinds of common synchronization problems. A "mux" can be setup to combine multiple types of data into a single virtual data type that can be processed more easily.

After processing its input data, a module usually outputs some result data back to the middleware for other modules to use as input. This is accomplished by using the middleware to construct a packet, attach some data and metadata, then pushing the packet back to the middleware. The delivery of these packets of data to other modules' Data functions is handled strictly by the middleware, so no additional communication logic by the programmer is necessary.

To port an existing piece of software, a programmer should first attempt to break apart the existing program into separable tasks of reasonable size. Required inputs and outputs between those tasks should be declared as data types and packets should be created for each data type. A module implementing each of the five necessary functions with some minimal setup and synchronization is easily created from a provided module skeleton (which consists of very few lines of code). Usually, the original code can be encapsulated inside a function, using the data packets provided to the Data function as inputs to the function. As

long as a program can be broken into parts, with the data required for each part clearly labeled, very little additional work is necessary for porting existing code (even serially executing code) to our middleware (which implicitly provides task parallelism). More complex handling or synchronization of data is still possible.

Programs are described by a configuration file, and optionally, additional arguments on the command line. The middleware initially loads a configuration file that describes the library of modules and configuration information for each module. This information is passed to the Register function for each module. The modules are automatically connected as described in the program graph construction method discussed. Some modules may be source modules – for example, a video camera or MPEG file reader may generate video frames that other modules can use to start a complex image processing chain. The configuration specified for an MPEG file reader module might simply be the input filename(s). For these special source modules that do not require any other inputs, the Data function may never be called; instead, a separate thread is created when Start is called, whose sole job is to push data packets. These data packets are pushed into the system where other modules that do need them as input can process them. Each module in a task graph continues with its processing until some final output(s) are created. This is repeated for each time quantum of data for the stream program until all pieces of data are processed. When no more data is available, the source module(s) report that they are finished, and each future module in the task graph finishes up with the data they have remaining. Eventually, each module's Stop and Shutdown is called, and the program exits.

3.2.3 Data Packets

The primary focus of our middleware is the processing of stream data. Although this is usually in the form of video, we do not restrict the type of data processed. The input data streams are split into separable, quantized packets of data. Each packet is assigned a timestamp to give its position in the stream. For video, this equates to individual images or frames of the video. Other data types, like audio, are split as closely as possible into relatable time quanta. For example, for a video frame packet that corresponds to the time interval 67-100 milliseconds (ms), the audio samples corresponding to the same time interval will be

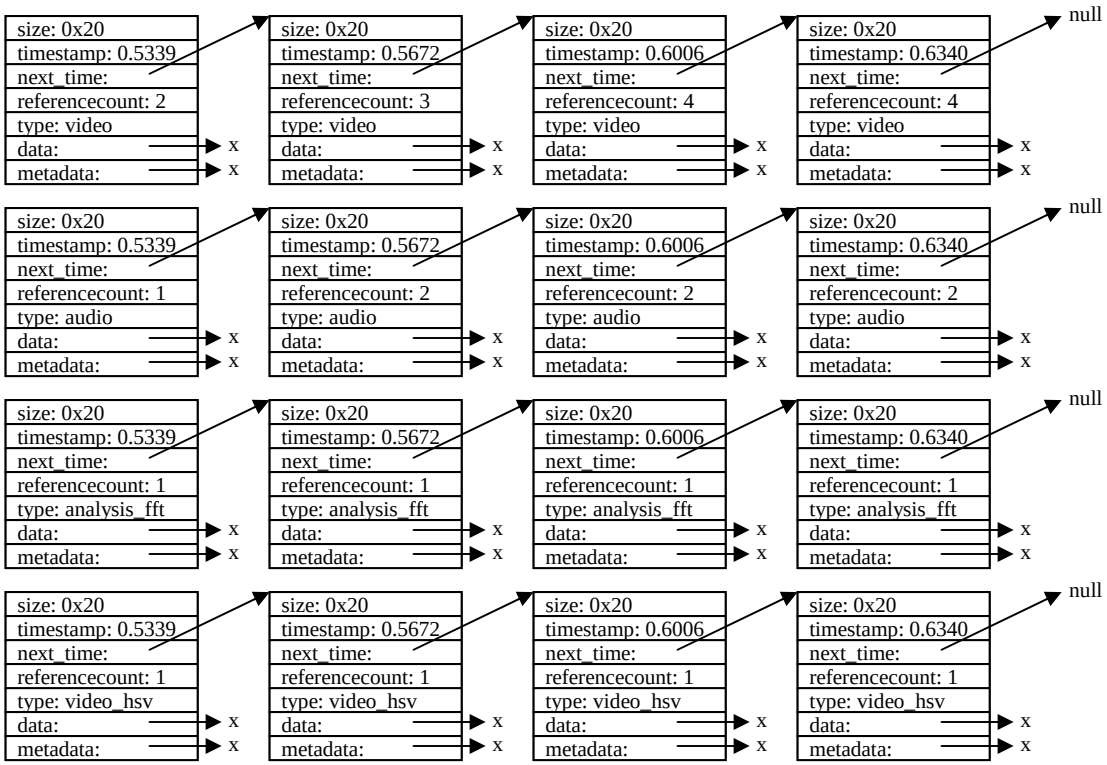

Figure 3.1: Organization of the packet system in the middleware. Packets are stored in a linked list, allowing both dynamic allocation and traversal to temporally close packets of similar type. Each type of data (e.g. VIDEO) is stored in its own linked list. The system maintains all the packets of the system through an array of linked lists (storing both the head and tail of each list), which each array index corresponding to a unique data type.

fully and exclusively contained in their own packet as well. The input stream for a particular type of data can be reconstructed by concatenating the packets of that same data type in chronological order. Different data types may be captured or processed at different frame rates. Because we consider video to be the most significant type of data (used often, difficult to split, but easy to logically quantize), we generally assign a reference time quantum based on a single, complete frame of video data. Then, we split other types of data based on the same corresponding time quantum. This makes it easier to synchronize across different data types; however, it is not a requirement of the system.

Each packet of data consists of some bookkeeping information for the packet itself, the actual data it contains, and some metadata describing that data. The actual format of this data and metadata is specified by the programmer and shared between tasks. For example, a video data packet would contain the video frame's pixel data, but also metadata in the form

of width, height, bit-depth, frame rate, etc. The packet's bookkeeping data consists of information such as the size, type of data, time quantum, reference counter, various pointers, and other necessary information. In ordered first-in first-out linked list (FIFO) is maintained for packets of similar type in order to form an asynchronous buffer of inputs and outputs between tasks. The linked list may grow or shrink as necessary, although imposing a maximum size will restrict the maximum memory usage of a program and also prevent one task from getting too far ahead of other tasks. The middleware can automatically decide how to optimally share or replicate (only if necessary) this data between tasks.

3.2.4 Communication

Modules do not communicate directly among each other, but, instead, by sending packets of data to the middleware. Each module registers the data types it wants to use as input and which data types it may output. When a data packet is output from one module, the middleware routes the packets of data from that module to the inputs of other modules that request the corresponding data type. This data is automatically buffered by the middleware until the modules that use the data can finish processing each packet of data. For each packet of data, a reference counter is incremented for each module needing the data so that the data can be freed once every module has finished using the data. If a module is too slow to process the data and the size of the buffer becomes too large, further buffering is temporarily blocked until buffer space becomes available. If a task cannot process data fast enough, the program will still run, but not in real-time. If buffer space never becomes available, the program will eventually stall, and the middleware will automatically alert the programmer of the situation, including which modules have stalled.

Because communication in this way is handled almost entirely by the middleware, modules never need to explicitly call any other module. When a packet of data becomes available for a module to use, the module is informed of this through a callback function and passed each packet of data one at a time. A benefit of this design is that each module can have task parallelism with every other module. Although there may be some dependencies between modules where parallelism is not possible, we can use this design to maximize task-parallelism of the overall program within a single time quantum and between time quanta. As

each module is essentially its own executable object, the middleware assigns a running thread to each module, giving rise to semi-automated multithreading. A well-designed program would consist of separation of many tasks (and thus many threads) in order to maximize parallelism. In addition, if desired, each module may create additional threads to handle data-parallelism. Alternatively, multiple instances of a module can be created to work on different parts of each data packet to handle data-parallelism.

Because each module is separated from other modules, we can view heterogeneous computing tasks as also separate modules. Indeed, a task running on (for example) a GPU will run in parallel with other code running on its host CPU. Similarly, a layout uploaded to an FPGA will run in parallel with its host CPU. Modules that are executed on different computers or different clusters of computers are all separated. The modularity exists as both a means to collaborative program design and to implicit parallel processing. For a large-scale project, modularity is already an important concept to maximize concurrent development time of modules.

The middleware abstractly handles all communication between modules and task allocation of those modules to the available computing architecture – the programmer need not worry about details more complex than the format of the data packets. If multiple modules want to use the same packet of data as input, the middleware optimally decides how to buffer and replicate the data as necessary. If multiple tasks on the same memory-sharing processor want to use the same packet of data, the data is not replicated, but instead the same pointer for that data is shared between tasks. Because of these communication and task abstractions, it is possible for the middleware to optimize many aspects of overall program execution with little to no effort on the part of the programmer.

3.2.5 Synchronization

There are two types of synchronization that occur: the synchronization of data packets within the system with modules that use them (middleware synchronization), and the synchronization of data packets that will be used within a module (application synchronization). For the middleware, care must be taken to ensure all data is handled in a thread-safe manner automatically. For the application, it is a different kind of

synchronization, in that the programmer (and thus module) may be interested in processing several packets of data simultaneously (e.g., video and audio with a matching timestamp). Because the Data function only receives one packet at a time, the application must synchronize the data across several calls of the Data function.

Application Synchronization

Although the application need not worry (for the most part) about thread safety with the middleware, the application must be able to synchronize the data for itself, which comes in one packet at a time, by properly combining it. The middleware provides functionality to aid in this. A multiplexer (mux) is a construct provided by the middleware for the logical grouping of data types. For example, if a module wants to be notified when a data packet is available for both audio and video, instead of registering for audio and video separately, the module registers with a mux that contains both audio and video with the AddInputMux function. Although the Data callback function still receives only single packets of data at a time, the module can call a middleware function MuxPacket containing the mux and the

```
    
    // mux is an allocated emMux object that manages the mux synchronization of data types and packets
    
1   int emmRegister (emModule* module, configPair* configuration) {
2       emcAddMux(module, mux, "VIDEO", 0, 0, 0, 0);      // add VIDEO type to mux
3       emcAddMux(module, mux, "INSIDE", 0, 0, 0, 0);     // add INSIDE type to mux
4       emcAddInputMux(module, mux);                      // add the mux to actual module's inputs
5   }
6   
7   int emmData (emModule* module, emPacket* pkt) {
8       emPacket* video;
9       emPacket* inside;
10  
11      if(!emcMuxPacket(module, mux, pkt, 1)) {   // wait for one packet of each data type
12          return 0;                              // else, simply break (process unfinished data later)
13      }
14  
15      emcGetMeta(video, "VIDEO", mux);     // fill 'video' packet with VIDEO data type from mux
16      emcGetMeta(inside, "INSIDE", mux);   // fill 'inside' packet with INSIDE data type from mux
17  
18          // process data packets...
19  }
```

Figure 3.2: Example module code for synchronization using a mux

incoming packet to determine whether all packets in the mux have been satsified (i.e., at least one packet of data is available for all data types within the mux). If the mux is satisfied, then the module could decide to process the logically grouped data as a whole. Any number of inputs may be included in a mux.

3.2.6 Feedback Loop

Although in related work, program graphs are typically represented directed acyclic graphs (DAGs), this may impose a limitation on the types of programs that could be used. We allow for program representation with cycles (or feedback loops) in the task graph. Although data will flow continuously as it becomes available with no particular intelligence associated with it (and no guarantee which module will get what data first), feedback loops are generally used by a module programmer already knowing what data type will be a part of that feedback loop. This data type and its feedback delay are especially important to inform to the middleware in the case of using a mux – at least one edge in the feedback loop cycle must have a delay attached to it in order for a mux to correctly synchronize a past piece of data with the current frame's data. The delay would indicate that the processing of the output from one task to the input of the next task would be processed in a future iteration of data in a stream program.

Although this feedback loop itself could provide a restriction on the amount of parallelism a program can achieve due to the circularly serial nature of the loop, it does allow the representation of programs that could not properly function without the cycle. Other components in the program can still function with full data and task parallelism as dependencies allow. To allow for maximal performance, the cycle time of the feedback loop should be minimized, as the cycle time of the loop can induce a minimum bound on processing time per iteration that would not normally limit a fully acyclic graph. The feedback loop delay is specified as an option to a mux input.

3.2.7 Data Filtering

A feature sometimes useful and desired by large modular programs and also by the aspect-oriented programming methodology is that of data interception. Explicit inputs and

outputs for each module are useful, but they may lack extensibility. For example, suppose we have a video capture module that captures video in its raw form. We also have a module that would like to receive the raw video data to do some image analysis. So, the video capture module outputs raw images and the image analysis reads the raw images. The system can easily match up all the inputs and outputs automatically (in this case, just one of each). However, this lacks versatility if the analysis module only accepts raw video data as input.

Suppose that the input video contains some confidential information. We want to disallow this confidential information from being passed on to the analysis component. However, the application developer might not have access to changing how the analysis module requests its input data type (for example, a binary-only module with no source code). Our solution to this situation is a middleware-provided method that allows other modules to intercept data at some point in the pipeline, do some modifications, and then push the same data type back out for other modules to use. We accomplish this through a prioritization of whether a module receives a data type before all other modules that utilize that data type. Communication and data processing proceed in order based on module priority if multiple modules request a data interception.

When registering for inputs and outputs, a module registers the desired data type for both input and output. Then, another function, SetPacketFilter, is also called during registration to indicate the priority of the data interception. When the middleware finalizes the registration stage, it reviews all packet filters to determine what priority each module uses for its outputs and assigns actual inputs from this set of specified priorities. So, following the example mentioned, a module could intercept the raw video data directly from the video capture module, modify it, and then write it back out before any other module was able to see the original video data. Modules that do not specify any priority will always receive the last (lowest priority) output of this data type. More detail is available in Chapter 4.

3.2.8 Profiling

Because our middleware acts somewhat as a supervisor for each task and each piece of data consumed and produced, we can automatically generate execution time statistics (profiling) for each and every module. We can use that information to find bottlenecks that

would be reported back to the programmer (where the programmer could decide to optimize the speed of a given module that was a clear bottleneck), automatically optimize the scheduling and allocation of tasks to processing nodes, determine if data is able to be processed in real-time, or vary the rate at which data should be processed to allow for real-time processing.

Although the middleware does not require real-time processing of data, online real-time analysis of video data is important for our case study application, so we provide this

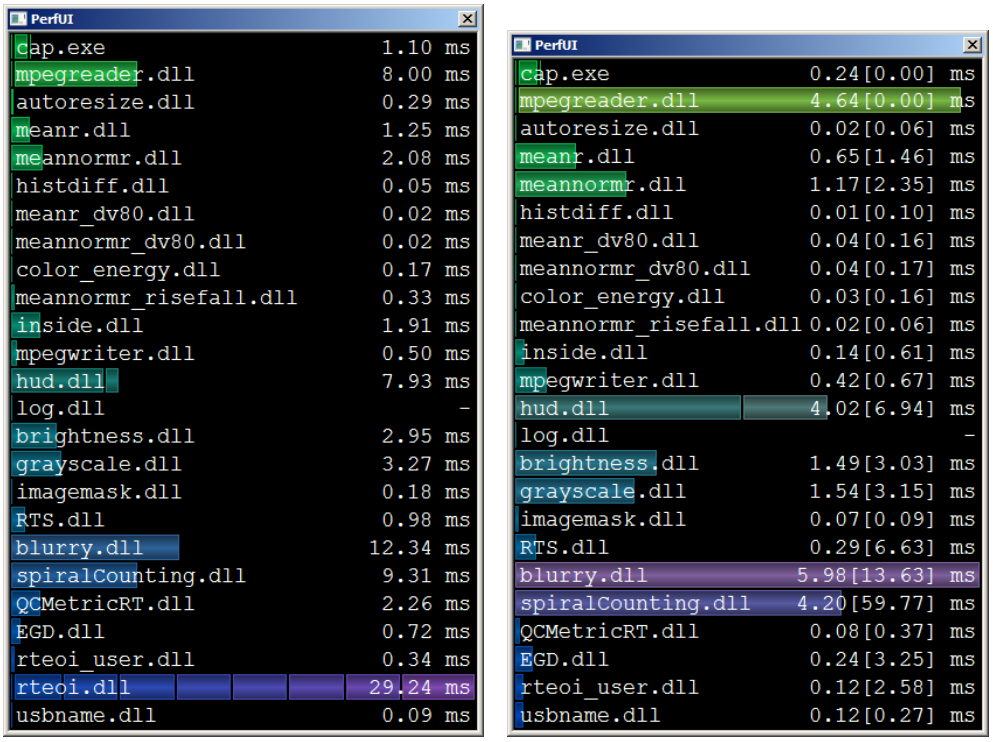

Figure 3.3: The real-time performance GUI of our middleware for two program configurations. (a) On the left, the average time per frame per module is reported. Each row represents a module within the system, and each individual bar represents the time each thread takes relative to other threads. We use multiple bars within a single module to represent the relative execution time of individual threads for that module. For example, RTEOI.dll takes about 29 ms, on average, total time across all threads. Because there are 8 main compute threads, each thread in takes about 3.5 ms on average. (b) On the right, the -perf option is used to get more accurate readings and also worst case frame time over the previous five seconds. For example, spiralCounting.dll only takes 4 ms on average, but in the worst case, it took nearly 60 ms to process a frame. This is because spiralCounting.dll skips some frames to be able to process without lagging behind. However, if there were a real-time deadline processing constraint of finishing each frame before the next one was received (33 ms for 29.97 fps video), then that module would not have been able to satisfy the deadline.

capability. An example of some of our real-time profiling capability is shown in Figure 3.3.

One problem with profiling and optimizing a program is that some components of a program may be data-dependent for how much computing time they take. For example, we know that some of our video may be completely black (e.g., when the endoscope is unplugged but video is continued to be captured). This video will generally be very fast to process, because there is no useful data in it. If we collect runtime profiles based on completely black video frames, our profiles will not be very useful. Similarly, there could exist some data that causes a module to take much longer to process than normal. A simple average of time taken per module might not be accurate for good scheduling.

We can collect this profile data automatically on a per-iteration level (i.e., each frame or data packet) per module so that we can gather not just average time per module, but also worst case, best case, and if the developer desires, every individual time to process every single iteration. For profile data collected on a per-iteration level, we can even construct a time-context-based schedule. When estimating the execution time of a program for scheduling purposes, a black frame might be processed fast by both modules A and B, whereas a non-black frame might be processed in some amount of time represented by some normal distribution. When estimating the execution time of a program in a specific iteration, instead of using only the average time, which could include both black and non-black frames, or a completely random value from some distribution of profile times, where module A could use a randomly chosen value derived from a black frame and module B used a randomly chosen value derived from a normal frame, it makes more sense to use a value derived from black frames for both A and B, or derived from normal frames for both A and B. This gives a more accurate estimation of the execution time for a program, and thus, more precise scheduling is possible.

3.2.9 Memory Leak Detection

In a complex program with significant multithreading, synchronization, and handling of data by many modules, it is easy to forget to deallocate some pieces of data. Although leaking the occasional piece of data might be insignificant in some programs, in a stream program, the same leak of memory may occur with every subsequent frame of data,

eventually causing the program to run out of memory. In order to help find these memory leaks, SAPPHIRE provides a built-in memory leak detection method. The OS's runtime memory allocation, reallocation, and free functions are hooked by overwriting the starts of these functions with a jump to our own functions. These functions then call their original counterparts, but while recording the inputs and outputs from these function calls, as well as the calling function's address. The result is a detailed map of the allocated memory regions, and what modules and functions called each individual allocation. Further details are described in Chapter 4.

Although data is continuously accumulated, it is not clear how leaks are detected from this data. Although we provide the memory allocation statistics through a profiling window updated in real-time which displays the memory taken for each module in the system, this is only useful for a programmer to observe absolute changes in allocation. For a more accurate and semi-automated method of memory leak detection, we heuristically determine leaks based on several factors: (1) ignore initialization allocations that will never deallocate by waiting a certain amount of time after the start of the program, (2) only memory that has been allocated for longer than about 30 seconds is a candidate for a memory leak (assuming about a 10 second buffering of data in the middleware), and (3) memory leaks are generally repetitive (considering that stream programs are repetitive), so when the same caller addresses occur repeatedly and frequently, they are more likely to be real leaks.

While (1) and (2) are automatically used to cull some memory allocations as legitimate, (3) is displayed by the middleware on-screen during execution time for the user to observe and determine in individual instances whether or not those allocations are real leaks. When the user sees many addresses that are identical and/or allocations that are very old, the programmer can use the function addresses to trace back to the source code exactly what line of code was causing the leak. Although this step requires some debugging knowledge, it is much better than not having the capability at all. Technical details are available in Chapter 4 and usage details in Appendix E.

3.2.10 Crash Reporting

For large multithreaded programs, it is sometimes difficult to debug and/or determine the cause of a program crash. So, we have a built-in feature to determine which module caused the crash as well as determining which functions cause the crash. This is useful for remote execution where a debugger is not available. However, the middleware also supports running under a local debugger to debug crashes without interference. For crash reports, the thread ID of the thread that caused the exception is available; developers can look through the list of loaded modules' thread IDs that have been kept track of in order to determine which module the crashing module actually is. We also have the exception address, but sometimes this is part of an unrelated library. More details as well as an example crash report are given in Chapter 4.

In debugging, we really prefer to find the crash address in the programmer's original module code. Often times the stack is still available in an exception, in which case, developers can walk the stack back examining various addresses at each stack frame. When one stack frame's return address matches that of a loaded module, the programmers have found the crash address in the original module. Some special exceptions are handled as special cases, such as statically linked C runtime or debug runtimes that are part of the loaded module but not part of the programmer's code. In these cases, the special cases are skipped until a better candidate for a crash address if found.

3.3 Common Modules and Data Types

Although SAPPHIRE is a completely generic framework, it is designed especially with stream programming in mind, and we intend to use it especially for our case study program, EM-Capture, a medical video analysis program. So, we have written several modules to assist in this usage. For example, a module that decodes data from video files and inserts the video frames into the middleware (mpegreader.dll), a similar module that writes the video frames back to a video file (mpegwriter.dll), a module that captures video from a video capture device (videocapture.dll), a module that displays the video data to the screen in its own window (hud.dll), and several others. All of these modules are highly configurable in order to suit a particular application's needs.

3.3.1 Video Data and Modules

Since we primarily work with medical video, the most important data type is the video frame. We declared a video data packet called VIDEO containing the pixel data, width, height, bits per pixel, frame rate, and data source. Several modules can produce this type of data, or a precursor to this data (called RAW_VIDEO), depending on the situation – mpegreader.dll to read a saved MPEG2 video file from disk, videocapture.dll to capture from a video capture device such as an endoscope, and screencapture.dll to capture a computer screen's display to use it as an input video stream. Because these are producers of data, they can be viewed as a source module in the parallel task graph. Each module offers a variety of configuration options.

Mpegreader.dll can take a single file, multiple files, directory of files, or a set of files specified by a filemask. The starting frame, ending frame or length of playback, and speed of playback can be set for each video. Videocapture.dll can capture from an installed video capture device that is visible to Windows and DirectShow. Most video capture devices include a driver to use with DirectShow, so this provides a simple common way to work with most video capture devices. The parameters for videocapture.dll include the device name, device input line (S-Video, Composite, etc.), width and height of the capture, width and height to scale the RAW_VIDEO output to, framerate, and bits per pixel. Screencapture.dll may be useful to feed the display of either the entire computer's desktop or a single window. This could be used, for example, as a remote desktop display, if the video data was sent over a network. Or, a video of what is happening on the computer could be saved to a video file. Configuration for this module includes window name (or left blank to capture the entire desktop), optional width and height to resize to, and framerate.

Several modules can also save or display the video data type, including mpegwriter.dll to save the video frames to an MPEG2 video file, aviwriter.dll to save the video frames to an AVI video file with any installed video codec on the system, getframes.dll to save individual frames to disk in single images in most common image formats, and HUD.dll (the heads-up display) to display the video frames to a window on the computer screen. In many situations, these modules are sink modules to the system, in that they might

not write any further packet data to the middleware, but they still process and output useful data somewhere else (e.g., to a file).

Mpegwriter.dll takes in its configuration the input data type (e.g., VIDEO), a filename format string that determines the output filename(s), encoding parameters like bitrate, quality, an optional "real-time" flag to ensure encoding never exceeds available CPU time, and several other optional parameters that configure how mpegwriter.dll might modify its behavior based on the length of the video (e.g., only keep videos that are at least 2 minutes long, write information about the file back to a data packet for the middleware, etc.). The filename format string is a string similar to something like sprintf that converts code strings into different strings based on some information about the video to be written. For example, %YYYY% is replaced by a 4-digit year (similar codes exist for other components of the date and time), %num% for video number (number of distinct videos that have been written so far by mpegwriter.dll), and %ip% for the IP address of the machine. This allows for distinct filenames to be used when multiple video files need to be written (for EM-Capture, we write one video file per medical procedure performed). Aviwriter.dll performs functionally the same as mpegwriter.dll, however it writes to an AVI video file instead of an MPEG2 video file. As such, the codec and its custom configuration is specified as another parameter and as a saved external configuration file. getframes.dll takes the file output type (e.g. JPG, PNG, GIF, BMP) as a configuration parameter.

Hud.dll is an important module in the system that allows us to see in real-time the video data and its results being processed while the system is running. This could be as simple as a video display (which only specifies the input data type and automatically displays it), or a complex synchronized display of video, some textual data representing the results of different modules' analyses, and even graphical feedback generated by those modules as an overlay on top of the video. This can be used in useful ways; for example, marking the location of a polyp during a colonoscopy. This module has numerous optional configuration parameters. To specify a text output of any packet in the system, a text= line is used in the configuration file. To display the width of the video frame, a line like "text=VIDEO:width/width: %d" would be used. The name of the data packet (VIDEO) and the name of the field within the data packet (width) is used to provide a source for the

displayed string. Then, a format string is specified for actually displaying the string on screen. This can be done any number of times and anchored to any corner or side of the screen in any font and color. To overlay graphical data (for example, to draw real-time feedback or non-textual information on top of the video), the overlay parameter specifies an input data packet type (which is formatted as common video data) and an overlay chroma key specifies how the video will be mixed (e.g., with transparency based on a color or alpha blending). A synchronization option allows data to either be displayed as soon as it becomes available, or to wait until all data for a particular frame is available so that they can all be displayed at once in a sychronized way. Synchronization is important for data and overlays to match up with the underlying video. Some additional properties include whether to display fullscreen or in a standard window, whether to record the combined hud.dll window's output to its own video stream (which could be combined with another module to write to a file), and some other special functions. Some interactive keys are available. For example, the '1' key hides all interface elements, the '2' key shows statistics about the middleware data streams (framerate and time to fully process each frame), and Alt-Enter causes the HUD to become a fullscreen window.

Emlive.dll is a variation on hud.dll that allows external programs to examine or display the video data running through SAPPHIRE in a synchronous manner. A shared memory space is setup to enable this.

Autoresize.dll is another commonly used module that can take some input video (usually RAW_VIDEO) and crop/resize it in order to provide a less redundant stream of video data. For EM-Capture, the video captured from the endoscopy hardware usually has a very large black border around the real video data, which could even be as small as half the captured video area. This module can automatically crop this input video to the smallest bounding rectangle representing the non-black area and write out a new data type (usually VIDEO) which is then used by the rest of the system analysis modules. This way, other modules do not need to manually crop/resize the video data themselves.

Imagemask.dll is a module similar to autoresize.dll, except that instead of cropping video, it instead generates a video stream in the form of a binary image mask of the input

video stream. Because the endoscopy video we receive may not be a simple rectangle, but rather an octagon or ellipse, it can be useful for other modules to receive this mask of "valid" video data rather than do this function on their own with potentially different implementations.

Videomixer.dll is a module that can be used to combine multiple video streams into a single video stream based on transparency or alpha blending. It is useful to combine several overlay video streams into a single video stream that can be passed to hud.dll.

Grayscale.dll is a simple module that generates a grayscale image of an input video stream. As several analysis modules may end up converting the RGB video data into grayscale for processing, it makes sense to put this into a separate module so that the operation is performed only once for all modules. This reduces redundancy as well as increases parallelism.

3.3.2 Simple Analysis Modules

The functionality of these modules is described in detail in Chapter 6. These are primarily used for endoscopic procedure detection for our EM-Capture program, but may also be used by other modules in different ways.

MeanR.dll takes video data as input, computes the mean red pixel value over the entire frame, and writes out a data type called MEANR. meannormr.dll does the same thing except computes the mean normalized-red pixel value and writes out a data type called MEANNORMR. histdiff.dll computes the motion of the input video stream between each successive frames and writes out a data type called HISTDIFF.

Meanr_dv80.dll and meannormr_dv80.dll each compute the variance of differences with outliers removed of the MEANR and MEANNORMR data types and writes out MEANR_DV80 and MEANNORMR_DV80. meannormr_risefall.dll computes how the MEANNORMR values rise and fall over time and writes several properties of this out as as MEANNORMR_RISEFALL. color_energy.dll computes the energy of various histograms generated over a window of time from the input video. This is another useful form of motion and scene detection. brightness.dll computes the average brightness of a frame of video.

Inside.dll combines all of the results of these simple analysis modules to determine where the start and ending frame of a procedure is from a continuous stream of images. A data type of INSIDE contains a flag of whether or not the corrsponding VIDEO frame is part of the same procedure or between procedures (outside-patient video frames, which should be discarded). A similar data type called INSIDE_NODELAY provides the same information except without a delay imposed on the INSIDE packet. The delay is required because inside.dll uses temporal features to determine whether or not a frame is part of a procedure. While INSIDE is delayed up to ten seconds (the duration is based on our inside detection algorithm) until a final decision can be made for the start a procedure, INSIDE_NODELAY gives an immediate conservative guess about whether the frame is part of a procedure. It will report false positives if it thinks a frame might end up becoming part of a procedure, but otherwise, it never reports false negatives.

3.3.3 Helper and Extension Modules

Log.dll provides a synchronized and combined logging system for modules to use. A module uses a function identical to printf (called eprintf, with some extra parameters that include the severity of the logging event, timestamp, etc.) in order to send information to log.dll. This function, in reality, creates a packet of type LOG that contains the necessary information. Log.dll then listens for these packets and can write them out one at a time to a logging file specified in the configuration file. Some additional parameters can be provided to this in order to remove non-severe logging messages, cycle logs, etc. Although modules can certainly implement their own logging functionality or simply print to screen, this module includes all the synchronization (where printing to the screen is a resource shared by all modules and may have multithreading issues) and output/severity/configuration options necessary for typical log files.

Tcp.dll converts SAPPHIRE from a single system middleware to a cluster-visible middleware. Some information is specified such as hostname and port to listen on and remote systems to connect with. Then, data types are specified that need to be transmitted to and from each host. Because individual modules never actually communication to other modules

in the system, some module that expects the VIDEO data type might get its VIDEO data from tcp.dll from a remote system rather than a local videocapture.dll.

3.4 Example EndoCapture.ini

SAPPHIRE programs are specified by configuration files. These files are simple text files that are a list of modules to load and their parameters. For example, Figure 3.4 is a configuration file specifying a program that shows the autocropped video of a capture device in a window on screen:

```
1  [videocapture.dll]
2  video.device=            # left blank to use any available device
3  video.mux=Composite      # use Composite input line
4  video.width=720          # input video width of 720
5  video.height=480         # input video height of 480
6
7  [autoresize.dll]
8  [hud.dll]
```

Figure 3.4: An example configuration file, specifying a SAPPHIRE program and configuration.

The '#' characters are used to insert comments. Some parameters can be left out to use the internal defaults for a given module. To change this program from capturing from a capture device to reading from a video file without changing anything else about how the program processes data, the videocapture.dll can be replaced by an mpegreader.dll with some parameter changes. The other modules' parameters can be left untouched. More complete examples can be found in the Appendix D. Modules can be disabled without removing them from the configuration file by inserting the comment character in front of the module name. The parameters listed under a commented out module will automatically be skipped as well.

3.4.1 Example Task Graph

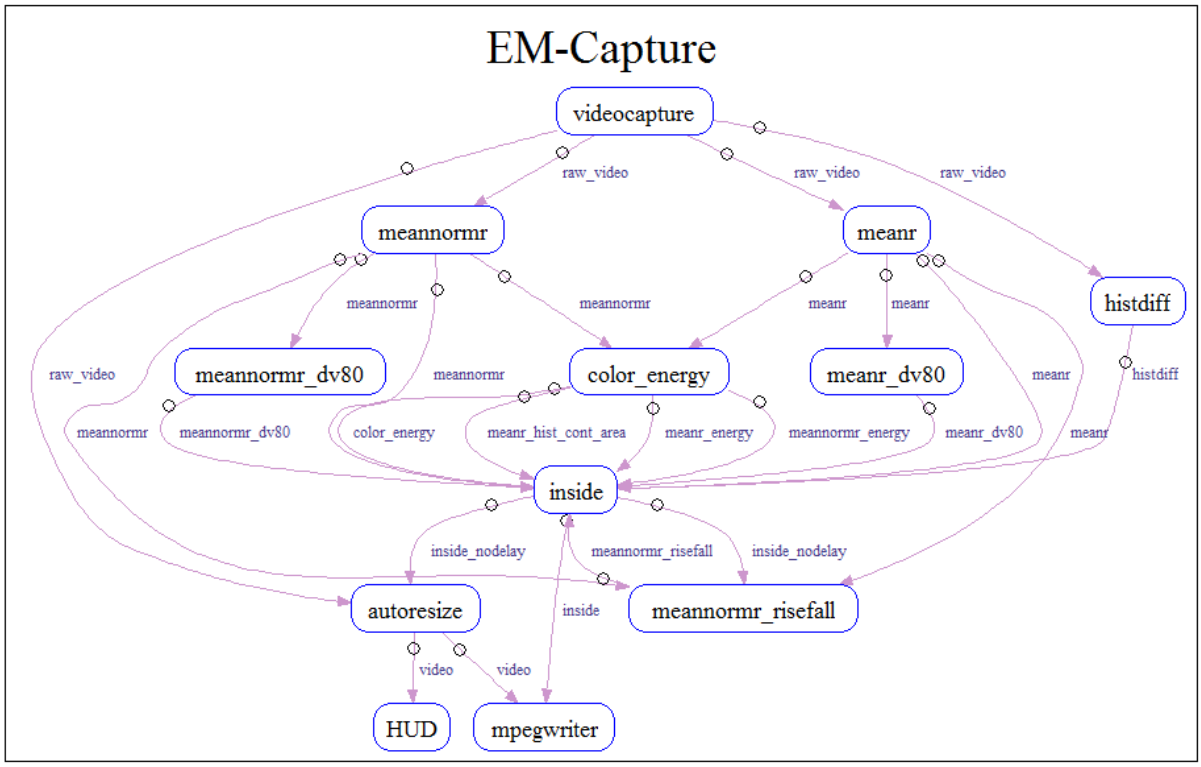

Figure 3.5: A task graph of the EM-Capture program written using SAPPHIRE. Each module is shown in a bubble while communications between modules are shown as directed edges. SAPPHIRE internally handles the communication, so in reality, the data flow is somewhat different than the task graph shows (same data types may be grouped so as to not incur multiple communication costs). However, the visual task graph is useful for determining data flow between modules, finding potential errors in design or missed producer modules, etc. For every program, working or not, SAPPHIRE automatically generates a .dot graph file similar to this that is readable and displayable by AT&T's GraphViz program.

CHAPTER 4. SAPPHIRE INTERNALS

Some of the internal details and usage examples for SAPPHIRE are provided. These give a more in-depth look at how SAPPHIRE works. Some functions and structures for our specific implementation of SAPPHIRE are prefixed by "em" (EndoMetric), "emc" (EndoMetric-core), or "emm" (EndoMetric-module).

4.1 Pseudocode for a Main Program

SAPPHIRE has a default program that parses and uses a configuration file which is just a static text formatted file. A simple pseudocode of the default program is provided. Recall that emmRegister(), emmStart(), emmShutdown(), emmData() are functions that developers need to implement as mentioned in Chapter 3.1.

4.2 Program and Virtual Graph Construction

Upon starting SAPPHIRE, the configuration file is loaded and parsed (Figure 4.1, Lines 1-7). Each module is loaded and its emmRegister method is called and configuration parameters passed in (Figure 4.1, Lines 8-10). Modules then register for various types of input and output data by calling the SDK's emcAddInput, emcAddOutput, and similar functions. All of the registration information is then accumulated in the middleware's internal state variables. Once all modules have been loaded and finished registering, the middleware takes sole control in order to determine the internal routing between modules and data types and starts all the worker threads by constructing a virtual bipartite graph of modules and data types and edges between them (Figure 4.1, Lines 12-21). The middleware waits until each module has no more data to process (Figure 4.1, Lines 23-25) and calls emmStop and emmShutdown to signal all the worker threads to stop and clean up (Figure 4.1, Lines 26-31).

```
    // G = {V, E}; a bipartite graph of a set of vertices V and a set of edges E
    // V = {V_m, V_t} where V_m is a set of module nodes and V_t is a set of data type nodes
    // M is a set of modules and their properties such as input and output data types
    // P is a set of their corresponding set of parameters
    // T is a set of data types

1   main() {
2       initialization();          // initialize M, P, and T to be empty sets and initialize other variables
3       parseProgramArguments();   // parse arguments from the command line
4       M, P ← parseInternalConfiguration();          // parse built-in modules and parameters
5       Mtemp, Ptemp ← parseConfigurationFile();      // parse modules specified in the .ini file
6       M ← M ∪ Mtemp;
7       P ← P ∪ Ptemp;                                // merge all modules into one set
8       foreach module m in M and their parameters p in P {
9           T ← T ∪ callEmmRegisterToGetDataTypes(m, p); // register all modules and get their data types
10      }
11          // replace modules' I/O types that have priorities other than default priorities defined
12      M, newTypes ← updateDataTypesWithPriorityOverrides(M, T);
13      T ← T ∪ newTypes;                             // add new override types to set T
14      G ← createVirtualDependencyGraph(M, T);       // create graph and prune graph
15      M, T ← getUpdatedUsefulModulesAndDataTypes(G); // update M and T from pruned graph
16      foreach module m in M {
17          createAThreadToRunModule(m, moduleThreadStart);
18              // moduleThreadStart is the generic module worker thread function
19              // threads are created in a suspended state
20      }
21      startAllThreads();                            // signal all worker threads to start
22          // worker threads call their own emmStart and emmData
23      while numberOfModulesStillProcessing() > 0 {  // loop until all modules have finished
24          wait;
25      }
26      foreach module m in M {
27          callEmmStop(m);      // after all modules are done processing, signal emmStop of each module
28      }
29      foreach module m in M {
30          callEmmShutdown(m); // after all modules have stopped, signal emmShutdown of each module
31      }
32  }
```

Figure 4.1: Pseudocode for the main program of SAPPHIRE. Complexity for registration itself has worst case running time of $O(m \cdot n^2)$ where m is the number of modules and n is the number of data types. Complexity for graph construction and other parts are discussed in their respective pseudocode listings. Complexity for subsequent execution of modules is O(1) per thread (not counting the actual processing that modules do), where m threads are created.

```
    // A thread is spawned starting at this function for each module in the system
    // m is the module that this thread was spawned for

1   moduleThreadStart(m) {
2       callEmmStart(m);
3           // perform work loop  (a more detailed work look is available in Section 4.3)
4       while m is not finished {
5           K ← findNewPacketsOfDataForModule(m, G, T);      // get a set of packets to work on
6           foreach packet k in K {
7               callEmmData(m, k);                           // process each packet
8           }
9           if allSourceDataTypesForModuleAreProcessedAndFinished(m) {        // if no more data,
10              setModuleFinished(m);                                          //   then set finished
11          }
12      }
13  }
```

Figure 4.2: Pseudocode for moduleThreadStart called by the main program in Figure 4.1.

4.2.1 Packet Filters and Priority Overrides

The first step in generating a virtual program graph is creating virtual data types if any packet filters or priorities are specified (Figure 4.1, Line 15). For example, if a module wants to modify or filter some data type, rather than just overwriting the original data type, a new virtual data type is created in its place. Although this virtual data type can point to the original data, it is internally treated as an entirely new data type. Then, other modules that use the filtered data are automatically switched over to using the new virtual data type.

In order to determine the new data types for the routing with priority overrides, a sorted list is created for each data type using the priority level as the sorting key. The actual priority level for a data type that each module uses is supplied through the emcSetPacketFilter function (Appendix A) as one of its arguments. If no level is set, then by default, the lowest level is used by consumer modules and the highest level is used for producer modules. Using the sorted list, discrete levels of priority are determined. For each successive level, the input data type for a module uses the level specified by the priority, and the output data type uses the next lowest level specified by any other module.

For example, consider the following situation. A videocapture module outputs a VIDEO packet as a producer with no specified priority. A HUD module displays the VIDEO

packet to screen with no specified priority. Then, two filter modules that use VIDEO packets as input are inserted, where filter1 has a high priority of 1000 and filter2 has a lower priority of 500. The highest priority for VIDEO in the system among all modules is 1000; thus, the original producer (videocapture) will now output "VIDEO:1000" as its data type. The filter1 module takes "VIDEO:1000" and outputs "VIDEO:500" (with 500 being the next highest priority). Then, filter2 receives "VIDEO:500" and outputs "VIDEO:0" which is shortened to "VIDEO". Because there are no other filter priorities specified by any other module, the filter2 module outputs "VIDEO:0"; however, a priority of zero is shortened to the original data type's name "VIDEO". The HUD module, not having specified any priority, then receives the lowest priority VIDEO packet, which is simply "VIDEO". If multiple modules use the same priority, then they will both receive packets for that priority, as would be consistent with the priority system not in place.

4.2.2 Generating the Virtual Parallel Task Graph

A virtual graph is created based on all the modules and their data types. This graph is a bipartite graph consisting of two sets of nodes: V_m for module nodes and V_t for data type nodes. Every module and data type is represented in this graph. The edges coming in to a module represents the data it will use as inputs; the edges going out of a module represents the data it writes to the system. The edges coming in to a data type represents the different modules that will output that data type; the edges going out of a data type represents the different modules that will use that data type as input. It is not possible for a module to be directly connected to another module or for a data type to be directly connected to another data type. Modules and data types are all matched up and relations are thus defined in this graph. Based on this information, we can then prune the virtual graph, which is then used for the internal routing. Although the virtual graph represents the data flow between modules, data is in fact never communicated directly between modules. Data is sent directly to the middleware, and only the middleware decides if and when to relay data.

```
    // G = {V, E}; a bipartite graph of a set of vertices V and a set of edges E
    // V = {V_m, V_t} where V_m is a set of module nodes and V_t is a set of data type nodes
    // M is a set of modules and their properties such as input and output data types
    // T is a set of data types

1   createVirtualDependencyGraph(M, T) {
2           // compute a list of modules that uses a data type as input, for each data type
3       foreach datatype t in T {
4           t.ConsumerModules ← Ø;
5           foreach module m in M {
6               If m takes t as input {
7                   t.ConsumerModules ← t.ConsumerModules ∪ m;
8               }
9           }
10      }
11          // compute a list of modules that produces a data type as output, for each data type
12      foreach datatype t in T {
13          t.ProducerModules ← Ø;
14          foreach module m in M {
15              if m takes t as output {
16                  t.ProducerModules ← t.ProducerModules ∪ m;
17              }
18          }
19      }
20          // construct the virtual graph
21      V_t ← Ø; V_m ← Ø;       // graph vertices consist of two sets: V_m for modules and V_t for data types
22      E ← Ø;          // edges are directed and represent the flow of data between V_m and V_t
23      foreach module m in M {
24          V_m ← V_m ∪ m;                      // add each module as a vertex
25      }
26      foreach datatype t in T {
27          V_t ← V_t ∪ t;                      // add each data type as a vertex
28          foreach module m in t.ConsumerModules {
29              E ← E ∪ newEdge(t, m);          // add edge from a data type t → a consumer module m
30          }
31          foreach module m in t.ProducerModules {
32              E ← E ∪ newEdge(m, t);          // add edge from a producer module m → a data type t
33          }
34      }
35      V = {V_t, V_m};
36      V, E ← pruneUnusedModulesAndDataTypes(V, E);   // prune unused modules and data types
38      G ← Graph(V, E);                        // create graph from V and E and return it
39      return G;
40  }
```

Figure 4.3: Pseudo-code for creating a bipartite graph representing the parallel task graph. Complexity for creating the lists of modules for data type inputs and outputs is bounded by $O(m \cdot n^2)$ where m is the number of modules and n is the number of data types. Then, constructing the virtual graph is O(1) for each output vertex and edge. Complexity for graph pruning could require a full graph traversal per vertex removed – $O(|E| \cdot |V|^2)$. Although these may be made more efficient, the sizes of these structures in practice are usually not large enough to focus on this aspect of the middleware.

4.2.3 Data and Module Pruning

Stray data types that have no output edges indicate that no modules want to use this data type as input. Thus, if this data type is ever sent to the middleware, it is immediately discarded. Furthermore, if all the outputs of a module are pruned in this way, the module itself does not output anything useful. Thus, it can be removed to save CPU and memory resources. This process is repeated over the graph until no more pruning is possible (Figure 4.3, Line 38). A module can override this pruning behavior by registering for a special output data type that is not pruned by the middleware. For example, a module that writes to a file rather than sending packets of data to the middleware might normally get pruned off. If it were pruned, the desired file would not be created. The module can register for the special output data type to indicate that it is a necessary module in the system and prevent it from being pruned. An example of this type of module is mpegwriter.dll (Section 3.1.1).

4.2.4 Updating the Internal State

Using the updated graph of modules and data types, the internal arrays and other variables of the middleware are initialized or updated. This state defines the final mapping and routing between modules and data types (Figure 4.3, Line 39).

4.2.5 Starting the Program

The modules then each have their emmStart method called. This allows modules to change behavior based on the modified internal routing of the system (for example, a module can check whether an output it was able to provide is even necessary – if it is not needed, the module can update its settings to no longer spend CPU time to create that output). Modules may also create new threads at this time if they want.

4.2.6 Data Processing

A work loop thread for each module is begun. The work loop is technically part of the middleware rather than each module. This does all the behind the scenes redundant checking of inputs and outputs being satisfied, thread-safe data routing, packet/memory management, and eventually, the calling of a module's emmData method. Additional bookkeeping and performance gathering code is also placed in the work loop to allow for automated profiling

and statistics of each module. Manual thread scheduling is also implemented in the work loop, if enabled, to allow or disallow threads from running on particular CPU cores (or at all).

4.3 Work Loop using Windows Scheduler

Each module (DLL) implements a method called emmData() which takes as input an emPacket pointer which represents a single data packet being passed to the module. From a programmer's point of view, this is a very easy way to receive a stream of data. However, internally, the middleware performs all the synchronization, setup, and calling of each emmData itself. For every module in the system, a thread is created. This thread works in an endless loop to process data as it becomes available. We refer to this as the work loop. Because the work loop is implemented by the middleware and not each module, not only does this ease the development process, it is also entirely possible for the SAPPHIRE core system developer to change the work loop to use a different scheduling algorithm or communication method. In fact, the entire internals of the system can be abstracted from the developer and entirely reimplemented with no change on the module programmers' parts as long as some infrastructure exists that calls emmData in the predefined way. After the middleware registers each module, a thread is created for each module, which in turn calls the emmStart for each module. The work loop of a thread for module *m* is shown in Figure 4.4.

Processing continues until all possible input data has been completely exhausted and the 'final' flag for each input data stream has been reached. This indicates that no more data will follow for that data stream for a given module. Once all processing has completed, the work loop is exited and a module's emmStop will be called.

```
    // G = {V, E}; a bipartite graph of a set of vertices V and a set of edges E
    // V = {V_m, V_t} where V_m is a set of module nodes and V_t is a set of data type nodes
    // K is an ordered list of packets (by timestamp in increasing order)
    // m is a single module assigned to the work loop
    // m.seenPacket[t] is the latest timestamp of packets of type t seen by module m
    // m.processedPacket[t] is the latest timestamp of processed packets by module m for type t
    // R is a set of profile data to be accumulated

1   R = Ø;                                  // set of profile data for this module initialized to empty
2   while not moduleFinished(m) {
3       BeginCriticalSection();                         // find packets that have not yet
4       K ← findNewDataPacketsForModule(m, G);          //   been passed to emmData for module m
5       EndCriticalSection();
6       if K ≠ Ø {
7           releaseTimeslice();         // if no data is available, yield CPU and try again later
8           continue;                   // go back to the beginning of the while loop
9       }
10      foreach packet k in K {                         // for each packet of data to process...
11          bindModuleAndScheduleProcessor(m);          // bind this thread to some CPU core(s)
12          startTime ← getTime();                      // record starting time for profile data
13          packetsToFree ← callEmmData(m, k);          // pass packet to emmData
14          endTime ← getTime();                        // record ending time for profile data
15          R ← R ∪ addProfileInformation(m, k, startTime, endTime);    // add profile data to set R
16          t ← getPacketType(k);
17          m.seenPacket[t] ← m.seenPacket[t] + 1;  // mark that m has had packet sent to emmData
18          foreach packet kFree in packetsToFree {     // for each packet to free or release,
19              tFree ← getPacketType(kFree);           //   mark that we have also processed it
20              m.processedPacket[tFree] ← m.processedPacket[tFree] + 1;
21              ReleasePacket(m, kFree);
22          }
23      }
24      if allSourceDataTypesForModuleAreProcessedAndFinished(m) {
25          setModuleFinished(m);           // if no more data, set as finished
26      }
27  }
```

Figure 4.4: Work loop of each thread. Finding new packets of data is accomplished by checking the middleware's internal latest timestamp of each packet pushed with the last seen packet of an individual module. This is a simple subtraction operation, and the module stores an absolute reference to index of the middleware's internal packet timestamps; thus this can be accomplished in $O(|V_t|)$ time. Then, emmData is called for each packet found. The emmData function itself can have its own complexity, but calling it and the surrounding code runs in constant time. The overall runtime across all passes through this loop (Lines 10-23) is entirely dependent on the number of packets processed (equal to number of iterations times $|V_t|$) and runs in constant time for each packet, thus this loop actually has amortized complexity of O(1) for each packet (and it does not execute unless it has a packet to process).

```
1  typedef struct emPacket_ {
2      int len;                          // length of this packet
3      char* type;                       // type of data
4      __int64 timestamp;                // timestamp of this packet
5      struct emPacket_* next_time;      // pointer to next attached data
6      struct emPacket_* prev_time;      // pointer to prev timestamp of same type packet
7      unsigned char* data;              // pointer to packet's metadata
8      unsigned int* meta;               // pointer to packet's attached timestamp of same type packet
9      int datalen;                      // length of raw attached data
10     int metalen;                      // length of raw attached metadata
11     struct emPacket_* parent;         // if this packet borrows the data from another packet, this points
                                            to the original data's packet
12     int PCR;                          // which index in PCRhead/tail this belongs to
13     unsigned int priority;            // filter/hook priority (output: 0 = normal/first generated, higher =
                                            hook first) -- used as a subtype
14     int final;                        // final packet of stream (typically blank for some types, but still
                                            valid; mainly used as a marker for cleanup)
15     struct emModule_* sourcemodule;       // source module that created this packet
16     int referencecount;               // reference count of this packet
17     int dummy;                        // ignore this packet
18     int channel;                      // for future use: multiple channels of the same type of packet
                                            that can have similar timestamps, etc; mpegreader and
                                            videocapture both working to produce RAW_VIDEO data
                                            type
19     __int64 PCRid;                    // like timestamp, but a unique id for every packet within one
                                            PCR (can have the same timestamps for multiple packets)
20     __int64 pushtime;                 // time of origination for this packet, from first pushed packet (in
                                            win32 performance counter)
21     void** releasecallbacks;          // array of callbacks when this packet is released
22     int numreleasecallbacks;          // number of callbacks
23 } emPacket;
```

Figure 4.5: Data packet structure

Because all work loops proceed in their own thread, the system is implicitly multithreaded based on the number of modules in the system. Each module can of course start their own threads to increase the amount of parallelism, but then they must manually handle their own synchronization since SAPPHIRE does not have any knowledge about the module programmer's multithreading implementation. By default, the worker (work loop) threads are scheduled by the operating system. It is possible to force threads to individual CPU cores and change thread priorities, however, this is very simple and does not necessarily provide any advantage over the built-in OS scheduler. SAPPHIRE has also implemented a custom scheduler for worker threads discussed in the user-mode task scheduler section in Chapter 5.

4.4 Data Structures

We describe key data structures we use for maintaining information about packets and the virtual task graph.

4.4.1 Packets

A simple example diagram of the packet linked list structure is shown in Figure 3.1. Some of these packet fields are not used directly by the user (programmer), but most are still used internally.

Most of these fields are self-explanatory, but the middleware itself does do some extra work beyond what the programmer has setup. When a packet is originally pushed to the system by a module, the middleware initializes and updates several fields. (1) The 'PCR' field (Figure 4.5, Line 12) is set to the internal array index for the corresponding data type. This provides a fast string name to type index (int) mapping for further use by the middleware (e.g., *PCRhead[pkt->PCR]*). (2) The 'type' field (Figure 4.5, Line 13) is updated to an internal static string so that modules never have issues about potential dynamic strings; also, this allows direct type == type comparison (if set up properly) which can be somewhat faster than the typical !strcmp(type, type) comparison. (3) The 'PCRid' field (Figure 4.5, Line 19) is set to a unique auto-incremented value based on the order it was pushed to give the packet a program-lifetime unique identifier. (4) The 'referencecount' field (Figure 4.5, Line 16) is initialized to the number of modules that will receive this packet. As packets are "released" by modules, the 'referencecount' field is decremented in a thread-safe manner. When the reference count reaches zero, it is only then actually deallocated from memory. (5) If the 'parent' field is set (Figure 4.5, Line 11), this means the packet shares data with that parent packet – the parent packet should not be released until the packet being pushed is ready to be released. This causes the parent's 'referencecount' to be incremented by one. When a packet is deallocated, its parent's 'referencecount' is decremented by one and also potentially freed if nothing else is using it. This is useful, for example, with the VIDEO data type, which crops the RAW_VIDEO data type without actually creating a copy of the video data (by simply using tricks with the video data pointer and stride). Since the VIDEO packet uses data directly from the RAW_VIDEO packet, it must set retain the RAW_VIDEO packet in

memory by setting it as a parent. (6) Other miscellaneous fields are updated or filled, such as 'pushtime' (the time the packet was pushed, for profiling) and 'sourcemodule' (the module that this packet originated from). (7) After all other fields are setup, the 'prev_time' field (Figure 4.5, Line 6) is set to the previous tail packet for this data type, the tail packet's 'next_time' (Figure 4.5, Line 5) is set to the new packet, and then the tail itself is updated to point to the new packet. Critical sections are used for some steps, but only when necessary.

This concludes the process of pushing a packet to the system. The system then maintains this internal state of packets and arrays of variables describing the data and modules of the system so that the work loop can determine whether data is available to process for each module. The work loop iterates through each input data type that the module listens for, and compares the most recent timestamps seen against the newest timestamps available for each data type. If there is new data, the work loop passes this data to the emmData function. The data passed in may be in any order between data types, but is always sequential within a data type. The work loop prefers to pass in data in a fair (round robin) fashion if it can, so that muxes can be satisfied as early as possible (as opposed to pushing several of the same data type that cannot satisfy a mux by itself). Due to the removal and abstraction of the work loop from each module's code, it is possible to change the scheduling and methodology of the work loop for all modules without actually changing any individual module's code.

Some modules take significant time and prefer not to produce data with every frame, or otherwise do not have meaningful data every frame. This can create a potential issue with modules that expect a timestamp-synchronized set of packets to process. In order to keep synchronization, producers should push packets for every timestamp (frame) even if they do not have useful data. This is done by pushing a packet with the 'dummy' flag set (Figure 4.5, Line 17), or by calling emcPushDummyPacket.

Some packets created by modules might have additional resources attached as pointers in the user data that have been allocated by a producer module. Eventually, this packet will be released by all modules and then freed by the middleware. However, because the middleware has no idea what the format of the user data is, it cannot know how to free

these attached resources. So, the middleware provides the function emcAddReleaseCallback to attach a custom resource destructor that will be called when the middleware is ready to free a packet.

4.4.2 Modules

Each module in SAPPHIRE is an emModule object (Figure 4.6), containing bookkeeping information about the module as well as function pointers to the implemented methods emmRegister, emmStart, emmData, emmStop, and emmShutdown.

Generally, the user never needs to use anything in this object (as it is almost entirely handled automatically by the middleware) except for the 'locals' variable (Figure 4.6, Line 2), which binds a module-instance-specific variables structure to a specific instance of a module. This allows a module to be loaded once but utilized multiple times for different kinds of configurations (e.g., potentially having multiple HUD module instances to display multiple video streams simultaneously). Because Windows will not load a second copy of an already-loaded DLL module, this is a necessity for the middleware to support multiple instances of modules.

4.4.3 Internal Arrays and Lists

A listing of the important internal variables and arrays and lists can be found in Appendix F.

4.4.4 Common Data Packet Formats

Some common data type formats are provided both for reference and as some simple examples. Module developers can specify their own data formats. The format should be shared with other module developers that plan to use the data. The middleware itself does not need to know the actual format of the data being communicated.

VIDEO

This format, shown in Figure 4.7, is a common video data format that is recognized by many SAPPHIRE modules. It can be extended to a new data type by appending new fields, while still being recognized by SAPPHIRE's provided modules.

```
1  typedef struct emModule_ {
2      void* locals;                    // local data for this instance of a module
3      int localsize;                   // size of locals; not necessary except for some functions
4      int moduleversion;               // version of this module
5      int (*emmRegister) (struct emModule_* module, configPair* configuration, struct
           emCallbacks_* callbacks);
6      int (*emmStart) (struct emModule_* module);
7      int (*emmData) (struct emModule_* module, emPacket* pkt, void* perf);
8      int (*emmStop) (struct emModule_* module);
9      int (*emmShutdown) (struct emModule_* module);
10     emPacket* inputs;                // array of input/output type definitions for this module
11     emPacket* outputs;               // the emPacket structure may only be partially defined as needed
12     int nInputs;                     // number of inputs/output types
13     int nOutputs;
14     __int64* nextpacketTS;           // timestamp of next unseen packet this module should process
15     __int64* lastpacketTS;           // timestamp of oldest saved packet this module retains
16     int* maxchainlen;                // maximum buffer length for a particular type
17     HINSTANCE dll;                   // Windows handle to this module
18     char* dllname;                   // filename of the module
19     HANDLE hthread;                  // handle of the main work thread for this module
20     unsigned int threadid;           // threadid of the main work thread
21     int finished;                    // flag to define whether this module is in the finished state
22     int nInputsNoDelay;              // number of non-feedback-loop inputs this module uses
23     int stopped;                     // flag to define whether this module is in the stopped state
24     int shutdown;                    // flag to define whether this module is in the shutdown state
25     int maxdelay;                    // maximum feedback-loop delay across all feedback delays
26     char* versionstring;             // version string for this module
27     int buildnumber;                 // SDK build version this module was compiled with
28     char* builddate;                 // build date string for this module
29     HANDLE mutex;                    // may be used to avoid async PushPacket, when multiple threads
           for the same module want to push
30     int nThreads;                    // thread local storage and profiling variables/arrays
31     HANDLE* allthreads;              // all thread handles for this module
32     int* allthreadids;               // all thread ID's for this module
33     __int64* threadtime;             // profiling for CPU for each thread
34     int* whichcore;                  // which core each thread is bound to
35     char* versionstring2;            // textual description and version of the module
36     char* description;
37     int internal;                    // whether this module is a built-in module or 3rd party
38     __int64 worstthreadtime;         // worst case thread time for this module
39     int processing;                  // whether this module is currently processing data or sleeping
40     int* memcount;                   // memory leak detection – number of allocations for each thread
41     __int64* memtotal;               // total memory allocated for each thread
42     __int64 memtotaltotal;           // overall memory usage across all threads for this module
43     int outstandingpackets;          // every allocated packet should be submitted to the middleware
           by a PushPacket, or else this variable continues to increment
44 } emModule;
45 emModule* allModules;                // list of all modules
46 int nModules;                        // count of number of modules
```

Figure 4.6: Structure that keeps information about a module

INSIDE

This data type, shown in Figure 4.8, is an example of a user data type. For the EM-Capture program, this is used to mark a frame as inside-patient or outside-patient. For each VIDEO packet, a corresponding INSIDE packet is generated. These two can be combined and synchronized with a mux for various purposes (e.g., only processing the VIDEO packet if the video frame is an inside-patient frame).

```
1  typedef struct meta_video_ {
2      int width, height;          // width/height of virtual video frame (may be different than
                                       width/height of original video frame)
3      int stride;                 // stride of video frame (same as original video frame)
4      int bpp;                    // bits per pixel -- usually 32 bpp for color or 8 bpp for grayscale
5      double framerate;           // set from source video -- usually 29.97
6      unsigned char* data;        // pointer to new data (the data pointer is the original video)
7      __int64 source_frame;       // frame number from video source (different sources may
                                       increment/reset this differently)
8      char* source_filename;      // 0 by default, but can point to an internal buffer for a source
                                       filename if available -- pointer not valid after packet is freed
9  } meta_video;
```

Figure 4.7: Structure of a VIDEO packet

```
1  typedef struct meta_inside_ {
2      int inside;          // 0=outside, 1=inside, 2=set upon transition from inside to outside
3      int frame;           // current frame number within a segment (all outside or all inside) of video
4  } meta_inside;
```

Figure 4.8: Structure of an INSIDE packet

4.5 Synchronization

Although many operations are designed to be implicitly thread safe, some operations do require the use of a mutex, which is used internally by the middleware. Most middleware API functions that modules call will take care of the thread safety issues.

4.5.1 Middleware Synchronization

Because significant amounts of data passing between modules and middleware is constantly happening, and no specific synchronization is done by the programmer, the

middleware undertakes full responsibility for this. The primary method of synchronization is with the use of a mutual exclusion object (mutex) in the form of a semaphore. Although the system has several of these mutex objects, the primary one that is used is for inserting and deleting packets. When a call is made to the middleware's PushPacket, the packet must be finalized and inserted into the middleware's linked lists of packets and internal arrays. Some finalization is performed, such as parsing the packet's data fields for correctness and consistency, and to prepare the packets for insertion into the system. For example, the type of the packet is converted from the type specified by the module into an internal type recognizable by the middleware; and, a reference count is assigned to the packet based on how many other modules intend to use the packet. Some profiling information is also setup at this time. If the number of packets pushed (for the type of data for that packet) has exceeded its maximum, then the middleware blocks the thread until other modules have caught up, and previous iterations' packets of that data type have been released.

Once this finalization is performed, a critical section is started by using the middleware's packet mutex object. The packet, middleware linked lists, and internal arrays are further updated by modifying pointers such as 'next_time' and 'prev_time' (next and previous packet in the linked list). A unique packet identification number is assigned to the packet. Some internal consistency checking is performed (to confirm that there will be no issues in the system once pushing this packet), and then the packet is finally "in" the system, ready for other modules to use it. The critical section is then ended so that the middleware can service other modules that call functions that will use this mutex object. For example, a subsequent release of a packet may require a critical section. To avoid unnecessary thread blocking, as much of the system as possible was designed to not need to use the mutex objects. When they are needed, multiple mutex objects are created for different purposes so that unrelated code does not block on each other.

4.5.2 Application Synchronization

The mux structure, shown in Figure 4.9, is updated each time a module calls emcMuxPacket. Most fields in the structure correspond to an array, with each type included in the mux representing one index in each array. Upon calling emcMuxPacket with a packet,

the corresponding index for the type of data packet is computed, and that index in 'count' is incremented by one, indicating that the mux has one more data packet available of that type. When all indices of 'count' have at least *n* packets available, a call of emcMuxPacket with a value of *n* will return indicating that the mux is satisfied. The module then knows that at least *n* packets of data are available and may be used. Upon finishing of processing, the module calls emcReleaseMuxPackets with a value of *n*, indicating that the module no longer needs *n* packets of each of the data types in the mux. Additionally, the 'count' array is decremented appropriately to indicate the number of remaining packets available for each data type.

The mux also supports optional data types. If a data type is not present in the overall system, then the mux will be satisfied even though that specific type of data does not have a 'count' of at least *n*. However, if the data type is present in the system, then 'count' must be satisfied. Thus, optional data types maintain synchronization, but only if they exist. If they do not exist, then the mux still functions by ignoring that data type.

For types indicated as a delayed packet due to a feedback loop, we initialize 'count' to 'delay'. When we check to see if a data type is satisfied in the mux, if we had a delay of one for a particular data type, then it means the very first set of packets satisfied by the mux at timestamp 0 will be satisfied without that delayed type since 'count' seems to be satisfied. Because 'count' is initialized only once, subsequent calls to emcMuxPacket will require actual packets to have been available. For both optional and delayed packets that do not have real packets available, null pointers will be returned for those data types, indicating that no data is available.

```
1  typedef struct emMux_ {
2      int nmux;          // number of types in this mux
3      char** type;       // type of data packet
4      int* optional;     // whether this particular type is optional
5      int* delay;        // feedback loop time delay (in iterations)
6      int* count;        // number of packets currently available for this type
7  } emMux;
```

Figure 4.9: Structure of the mux (multiplexer object)

4.6 Runtime Profiling

In order to support profiling, the work loop surrounds its call to emmData with a timer start and stop function (emcPerfStartClock and emcPerfStopClock). The time difference is computed and used as that module's execution time for each piece of data. While this handles most cases automatically, there may be the case where the programmer puts the worker thread to sleep manually while it waits for other data to become available (e.g., spawned threads to take advantage of multithreading). In some cases, this is automatically detected, by hooking some Windows calls such as CreateThread and WaitForMultipleObjects. However, it is better for the programmer to call emcPerfStopClock manually before using these artificial sleep methods and then start the timer back up with emcPerfStartClock after resuming. For the spawned threads themselves, although the Windows CreateThread function is hooked by SAPPHIRE in order to automatically detect newly spawned threads, it is still better for the programmer to manually call the start and stop functions to also do accurate profiling, as the built-in Windows GetThreadTimes function is only accurate when a thread expends a full timeslice (otherwise, the used time is not added to a thread's execution time) [70]. To retrieve an accurate timestamp, we use the CPU instruction 'rdtsc', which reads the monotonically increasing timestamp counter from the CPU. This generally increases at the rate of the processor's base clock speed per second (e.g., a 3GHz processor will increase by three billion in one second, with very high resolution, potentially incrementing by one for each clock cycle). SAPPHIRE prefers to use the high resolution timestamp counter whenever it is possible to encapsulate threads' code (e.g., through emmData or manual calls to the emcPerf functions). When it is not possible, SAPPHIRE falls back to using GetThreadTimes.

The profiling window (shown in Figure 3.3) can be brought up at any time during execution by pressing a key combination (Ctrl+Shift+Alt+F). In the normal running mode of SAPPHIRE, a lower overhead method of performance gathering is implemented. In this mode, only the average time per frame (over the course of the entire program execution up to the present time) for each module is displayed, along with the distribution of time among threads. By running SAPPHIRE with the '-perf' option, some enhanced performance gathering is enabled. One additional statistic is displayed, which is the worst case runtime per

frame for each module (which is reset every five seconds to prevent one long iteration of processing from preventing recent and useful information being displayed). This is useful for determining whether the worst case performance of a module is acceptable. For modules that normally process at a low frame rate, the average time per frame may seem reasonable, but the worst case time for processing an individual frame might actually be longer than desired (e.g., for deadline constraints).

In the enhanced performance gathering mode, the scheduling of threads is changed to "realtime" priority so that other threads cannot interrupt any module's emmData function. While context switches by the OS would normally interrupt threads and increase the observed execution time, this enhanced mode effectively disables context switching, increasing the accuracy of the gathering the performance statistics. Additionally, all threads are scheduled on a single processor such that they cannot interrupt or influence each other due to various factors such as Hyperthreading or TurboBoost. With Hyperthreading, two virtual processor cores are created for each physical CPU core. When one thread is run on each virtual core, they may in fact be running on the same physical core, competing for a single computing resource. This, in turn, may extend the execution time for tasks running on hyperthreaded cores. For TurboBoost, a computer that utilizes only a single core may have its processing speed greatly increased beyond the listed processing speed. However, when using multiple cores, this boost in speed decreases based on the number of cores being used. When all cores in a system are being used, the processing speed may drop fully back to the manufacturer's listed processing speed for a processor. By using only a single core, we attempt to avoid the varying of CPU speed from TurboBoost.

4.7 Memory Leak Detection

Our middleware hooks the Rtl* functions in ntdll.dll associated with memory allocation (e.g., RtlAllocateHeap, RtlReAllocateHeap, RtlFreeHeap). These functions are the lowest level Windows runtime library associated with heap allocations of non-page sizes; these are eventually used (after several layers of abstraction) by C malloc and C++ new. The hooking is done by getting the function address of each function with GetProcAddress, inserting a jump opcode at the start of each function to go to our own hooked function,

running some of our own code in preparation for calling the original function, calling the original function, running our own code again after the original function returns, then finally returning control back to the user program. As nearly all allocations (including those from 3rd party libraries) go through these Rtl functions, we are able to effectively hook all memory allocation a program does without needing to recompile or modify any source code.

Each time one of our hooked functions is called, we walk the stack of the calling thread in order to determine where the allocation originally occurred. The stack saves the return address of every function call. While walking the stack, we generally encounter several different layers of abstractions in libraries, which we usually can ignore. For example, a C malloc call may jump from a user module's DLL, to the C runtime msvcrt.dll, to the kernel32.dll HeapAlloc, and finally to the ntdll.dll RtlAllocateHeap. There may be several stack frames defined in each of these successive modules, but we primarily want to find the original user's module. We can find which module owns which range of code addresses by enumerating the modules and their regions of memory with the Windows function CreateToolhelp32Snapshot using the TH32CS_SNAPMODULE subfunction. We continue walking the stack until we reach an address that is part of the user's module. This gives us the return address in the user's module, which we can convert back into the original line of source code if need be, either through a debugger or manually. Detailed usage information for the memory leak detection can be found in the Appendix E.

4.8 Crash Reporting

While crash reports and debugging may require advanced system knowledge, we provide some enhanced information to make this somewhat easier, especially for debugging on remote systems, where a debugger might not be installed. An example crash report is shown in Figure 4.10.

If run on a local system with a debugger, similar information would be seen. However, it may be difficult to determine the module that caused the crash, especially if the crash address occured in a system or third-party DLL file. Because SAPPHIRE supervises all of its threads, it knows which thread belongs to which module. So, the middleware can immediately determine the offending module.

For a crash involving a trashing of the stack, it may be impossible to get any information from a stack trace, whereas knowing which module the thread belongs to would be an important start (especially in a multithreaded system involving dozens of threads). If the stack is intact, a full stack trace will be provided along with the module that owns each stack frame. Although symbols are not immediately available from the crash report, some common compiler tools such as dumpbin may be used to find what functions the addresses in a stack trace belong to, providing some additional help for debugging.

For remote systems where a debugger is not available, the details of the crash report are conveniently saved to a file on disk, which can then be sent back to the respective developer. With some knowledge, it may be possible to use the crash report to trace back the crash to the offending code.

To support crash detection in a way that allows both catching the crash and still allow SAPPHIRE to be run under a local debugger normally, we use the Windows SetUnhandledExceptionFilter function to set the application's global crash handler. Although other methods of exception handling exist, they were unable to provide friendly behavior to both catching the exceptions and using a debugger to debug the exceptions.

```
1   --- crashed ---
2
3       exception code: c0000005
4       exception address: 7746e582
5       exception thread: 1910 (hud.dll)
6
7       eax=000004f6  ebx=00000000  ecx=76170958  edx=00000000
8       esp=1acdf1b8  ebp=1acdf1cc  esi=000004f6  edi=000004f2
9
10      code bytes (ntdll.dll+0001e582) : f0 0f ba 30 00 0f 83 e4 ca 00 00 64 ...
11
12      stack frame 1acdf1cc :  next: 1acdf1d8 761172d9  cur: ntdll.dll+0001e582
13      stack frame 1acdf1d8 :  next: 1acdf210 760e4697  cur: msvcrt.dll+000472d9
14      stack frame 1acdf210 :  next: 1acdf72c 066e3430  cur: msvcrt.dll+00014697
15      stack frame 1acdf72c :  next: 04c18d68 0040476a  cur: hud.dll+00003430
16
17  --- end crash report ---
```

Figure 4.10: Crash report of the hud.dll module crashing when a bug is intentionally introduced

4.9 Video Processing Considerations

Working with video data at a full frame rate of about 30 frames per second in real time can require a great deal of processing power. Although some high level functions may exist (e.g., GetPixel(image,x,y) to get the pixel value of image at the specified x,y coordinates), these are rarely efficient ways to access the image data. In order to maximize throughput, it is necessary to minimize overhead to functions like this. Instead, we should access the data directly. Images usually consist of a pointer to the video data, width, height, bits per pixel, and stride. Although the meaning of most of these are obvious, the stride is a lesser known term. The stride refers to the number of bytes between rows. Usually, this is equal to the width (number of pixels in a row) times the number of bytes per pixel (bits per pixel divided by 8); however, this is not always the case. For example, the cropping module (autoresize) in SAPPHIRE that creates a cropped image based on the non-black visible region does not create a new copy of the video data, as this would cost a lot of time and memory. Instead, the cropping module refers to the original video data as a parent packet, and then creates a new video data pointer at the top-left of the cropped position within the image. The width and height are modified to reflect the cropped video data in a new metadata packet. The stride, however, remains the same as in the original parent packet, since its video memory has not changed.

To address a pixel directly, the address is calculated as (metavideo->data + y*metavideo->stride + x*metavideo->bpp/8). For an 8-bit image (e.g., grayscale), this is cast to an (unsigned char*) and read in as a single byte. For a 32-bit image (which most of SAPPHIRE's video data is), this can be cast to an (unsigned int*) and read in all the channels in the pixel at once. The 32-bit video data is stored in BGRA format, such that the lowest byte is blue, next lowest byte is green, and next byte is red.

Depending on the algorithm, as a point of optimization, it may be more optimal to calculate the address of the start of the row only once, and then iteratively read each pixel in a row without recalculating the address from scratch. This reduces the amount of pointer arithmetic. For simple video operations where very little math is actually performed at each pixel, this can greatly increase the performance. The key concept is that there are simply a

huge number of pixels – for example, 720 x 480 x 30 = 10 million pixels per second. Spending even an extra 10 clock cycles (which is a very small amount) per pixel could equate to 100 million clock cycles, which could be several percent of overall CPU usage. It is easy to see how even trivial operations at each pixel data point can be multiplied into a significant costs. This is exacerbated for high-definition video, where 1920 x 1080 x 30 = 62 million pixels per second. For simple operations, the overhead of recalculating the address could even exceed the cost of the actual video analysis. Loop unrolling by processing several pixels at once may also be used to reduce overhead costs – the autoresize module always ensures that video widths are a multiple of 8, as that is a requirement for the mpegwriter module. So, in many cases, processing pixels in multiples of 8 as well is a good idea to improve overall performance. Many of the common core components are optimized using x86 assembly code to provide extremely fast processing for the required or common overhead modules that would otherwise already exceed the processing power for a typical computer workstation.

4.10 Evaluation of SAPPHIRE

It is difficult to evaluate the correctness of every aspect of such a large project. We have attempted to do this through design review, case study implementation, consistency checking, stress testing, crash detection, and error reporting.

4.10.1 Case Ctudy Implementation

To show that our middleware is able to function as designed and provide a robust environment for programming, we implemented two case study programs. One program was a port of an existing video analysis program, EM-Capture, to use SAPPHIRE, while another was an extension of EM-Capture, to perform complex analyses in and provide real-time feedback to physicians in a real clinical setting. Both of these experiments have been successful in their goals and in showing that SAPPHIRE could reliably support them. These case studies are discussed in greater detail in Sections 6.9 through 6.11.

4.10.2 Consistency Checking

A special option when used to start SAPPHIRE, -check, will cause the middleware to enter a testing mode. SAPPHIRE will continue to function as normal, except that various built-in testing and debugging code will be executed at certain times in order to validate some operations and check the consistency of the internal state of the middleware. This is in addition to enforced consistency checking that is always turned on.

For example, when a module pushes a packet to the system by calling emcPushPacket, this normally involved several steps, where eventually, the packet is added to the tail of a linked list of packets. When -check is used and emcPushPacket is called, the internal state of the packet linked list is first checked for errors. If any errors are found, a report is displayed as such. Then, the packet is pushed. The state of the linked list is again check for consistency. Consistency checking is also done at several other times, such as packet release, validating the continuity of modules' packets' timestamps, etc. This automated checking has successfully found several bugs at various stages of development, both as part of the middleware and for module developers. As a result, we have a much more robust system.

4.10.3 Stress Testing

An option is available with the mpegreader.dll module that will cause the source video stream to enter a stress test mode. In this mode, while the video(s) specified will still be played, the size of the video is constantly varied to simulate rapidly changing conditions that could rarely occur in practice for our case study (e.g., endoscope being unplugged but still giving a partial video frame). As many modules in our system also depend on whether the video is part of a procedure or not, the video stream is also blacked out often in order to simulate entrance and exit segments. This causes several modules to switch states back and forth between inside and outside, faking the occurance of many procedures in a short period of time. This stress testing mode has found numerous bugs, and has also been useful for tracking down bugs that occur too slowly to notice normally due to the occurances of those bugs not happening often (such as leaks of memory when a new procedure happens).

4.10.4 Error Reporting

As we have run SAPPHIRE in a development environment for several years (since 2008) and in a live clinical environment since 2010, we have had many opportunities to encounter errors, from developers implementing modules, from testing, from physicians, and from our automated error reporting systems. The combination and variety of sources for errors to be discovered and feedback in general has increased the robustness of our platform.

4.11 Summary and Future Work

SAPPHIRE supports a wide variety of features expected from multiprocessing middleware while simultaneously making the development process much simpler compared to traditional programming practices and other existing work. SAPPHIRE supports a wide variety of features. However, it is possible that in the future, some unforeseen features would be requested by developers that could require changes to the middleware. The modular design of the middleware has, in the past, made adding new features simple both for the maintainer of the middleware and for the module developer.

The middleware does provide some common built-in modules, however, there are still relatively few modules compared to the huge libraries that exist across the internet. Although most of these libraries can be used within SAPPHIRE with additional programming on the part of module developers, it would be much simplier if they were able to use them as built-in modules. For example, to encrypt data, an encryption library would need to be interfaced with (or code developed) by the developers themselves. The developer might choose to simply include the calls to encryption in their own module (serial) or create a new module that supports some data types that could be used to parallelize the encryption process. It would obviously be better if these common types of libraries were built-in to SAPPHIRE to reduce the work for developers, and improve the efficiency of programs by ensuring the libraries are implemented with proper parallel task design.

While SAPPHIRE has been primarily designed and tested for local system development (e.g. a single computer and attached devices, such as a GPU), we would like to have enhanced support for cluster, grid, and cloud computing. Although we do have some capabilities to handle this, it would not be as simple for a developer to use cluster resources

for multiprocessing in SAPPHIRE as it currently is to use SAPPHIRE for a multicore computer.

Although SAPPHIRE has a default program that parses and uses a configuration file which is just a static text formatted file, it would also possible to change this to something graphically configured, allowing a graphical specification of a program. Also, instead of static modules that are only loaded at program start, it would be possible to add support for dynamic loading/unloading of modules at runtime (e.g., if users' needs or specifications changed during runtime, but the system cannot be brought down due to some mission critical software concerns).

CHAPTER 5. TASK SCHEDULING OF STREAM PROGRAMS ON HETEROGENEOUS SYSTEMS

In this chapter, we present a problem formulation of task scheduling for stream programs on heterogeneous systems such as workstations with multicore CPUs and/or Graphical Processing Units (GPUs). Examples of stream programs are analysis of images during a medical procedure for computer-assisted surgery or computer-aided screening. Next, we present a heuristic algorithm for task-to-processor allocation that assigns tasks to processors and a scheduling algorithm that determines schedules of the tasks assigned to a set of processors. We report the evaluation results of our algorithms compared to the closest related work.

Task scheduling for non-stream programs has long been studied. The parallel task graph (PTG), as shown in Figure 5.1, is a commonly used data structure to represent the tasks to perform (nodes) and communication between tasks (edges that also define dependencies between tasks). The makespan is often the main performance metric. Consider all paths in a PTG from the source node to the sink node in the graph. The length of a path is the sum of the cost of all the nodes and the edges in the path. The *critical path* from one source node to one sink node is the length of the longest path among all the paths between these two nodes. The length of the critical path from start (source) to end (sink) for an overall program is called the *makespan*. The scheduling of a particular task is not just based on when all of its predecessors have completed, but also when there is available processing time on some processor. The step of deciding which tasks to run on which processors is called task allocation or task-to-processor allocation. In a system with a limited number of processors (less than the number of tasks), task scheduling is an NP-Complete problem [22]. Thus, many heuristic algorithms were proposed for homogeneous and distributed/networked heterogeneous systems. The heterogeneity was typically in terms of CPU processing speed.

For stream program scheduling, the problem is even more complex, as each task must be run potentially millions of iterations over a stream of incoming data items. When heterogeneous processors (such as GPUs with CPUs) are used, the problem becomes more

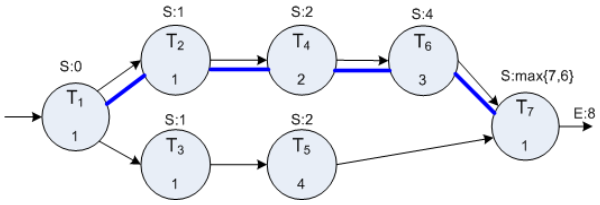

Figure 5.1: Seven tasks (T_1, …, T_7) are shown in this PTG with T_1 as the source node and T_7 as the sink node. Each node has associated cost (execution time). Edge has no cost in this example. *S* on top of each node denotes the earliest start time of the node. *E* denotes the output time of a given input. Task T_7 cannot start until both of its inputs are ready. Hence, the earliest start time of T_7 is the maximum end time of its input nodes T_6 and T_5 or max{7,6}, respectively. Tracing back from T_6 repeatedly in a similar way to the source node gives all the nodes along the critical path: $T_1 \rightarrow T_2 \rightarrow T_4 \rightarrow T_6 \rightarrow T_7$. The length of this path is 8 shown as the value of *E*.

difficult since choosing one architecture over another for a particular task adds another level of complexity. Differing amounts of speed up can be obtained from different tasks by using an implementation that utilizes GPUs over one that does not. For example, while one task may run twice as fast by using GPUs, a different task may run four times as fast. The exact speed up cannot be easily uniformly estimated, and existing work has only been done by normalizing different CPU speeds to the slowest or a reference CPU speed [25].

Task scheduling is a well-studied field with decades of related work. We discuss only the most important concepts and results that are related to our work and summarize most relevant existing work in Chapter 2.

5.1 Our Contributions

We develop a new static task-scheduling framework for stream programs on a heterogeneous multiprocessor system defined as a computer system with one or more multicore CPU and/or one or more GPU. Our framework does not unroll an original PTG as in a recent existing work [22]. Partial graph unrolling of the original PTG increases the graph size. For real applications with 10 to 100 tasks, the unrolled graph quickly becomes very complex. Most importantly, our framework supports *heterogeneous implementation (HIT) choices* where one task may have several implementations: CPU multithreading, CPU-GPU, etc. It is not uncommon to have several implementations, especially for utilizing GPU, since software development is often done iteratively to improve the speed of the execution. Running all GPU capable tasks on GPUs may not yield the best performance since they all

compete for the same computing resource (a limited number of GPUs). Our framework automatically chooses the best configuration (which implementation, which processors, and which time to execute the task) for each task. To the best of our knowledge, we have not found any existing task scheduling that supports HIT.

The framework has two major steps. (1) Task allocation that assigns tasks to processors (CPU and/or GPU). This step is optional, but provides a significant benefit to the next step. (2) Task scheduling determines schedules of the tasks assigned to a set of processors. Given many tasks in a stream program that runs over thousands or millions of inputs, our framework reduces tedious and complicated work for programmers to manually assign tasks to processors to achieve optimal performance.

We formulate the task allocation problem with HIT support as a load balancing problem that optimizes the maximum load (execution time) among all the processors in the system. Given large inputs, we proved that this maximum load approximates makespan of a stream program to within a negligible amount of error. This problem formulation enables us to have a simple heuristic load balancing algorithm called *K-HIT* that solves the formulated problem.

For the final task scheduling, we develop a variant of earliest finished time first for stream programs called *Stream-EFT*. We evaluate the performance of our K-HIT algorithm and Stream-EFT algorithm against cyclic scheduling as well as partial unrolling [22][42]. The simulation results show that our approach outperforms the existing algorithms in terms of the makespan of a stream program, while maintaining reasonable time and memory requirements.

5.2 Drawbacks of Related Work and Features of Our Work

Our task-scheduling framework does not require graph unrolling for the most computationally intensive parts, to avoid significant increase in complexity of the problem as aforementioned. Unrolling can be especially costly for algorithms that run in higher exponent polynomial time or exponential time. Writing GPU code for many tasks of a program and then running all GPU capable tasks on GPUs may not yield the best performance since they

all compete for the same computing resource (a limited number of GPUs). To consider whether it is better to run a particular task on GPU versus CPU, at least two versions of a task must be available: one that uses only CPU(s) and another that can utilize GPU(s). For GPU tasks, it may already be the case for two versions of a task to be available, since typically a CPU version is written for prototyping and testing before a GPU version is written for improved speed. When more than one version of a task is available and only one needs to be executed, we call this *heterogeneous choice*. This is a feature of our proposed work that is not discussed by the related work. Another common restriction in related work is that once a task is scheduled on a processing node, it runs until completion without preemption. It may not always be necessary in practice to keep this restriction, as modern processors have hardware preemption available. In addition, this restriction often prevents some related work from yielding an optimal solution. While some variations of cyclic scheduling may allow preemption within a single iteration [42], it does not allow for preemption between iterations, requiring unrolling and thus a blow-up of problem size. The related work in partial graph unrolling did not allow for preemption. Our proposed method allows for preemption in the form of *bubble filling*, where we attempt to fill unused periods of processor time (bubbles) with tasks that have yet to be scheduled.

It should be noted that the task scheduling preemption we refer to differs from typical OS task scheduling in that OS task scheduling generally tries to be "fair" by giving each task a time-slice in round-robin fashion. Each task is continuously preempted by other tasks, effectively causing a worst case scenario of end times for tasks. Because future dependents cannot run until their dependencies have been satisfied (i.e., those tasks have reached their end time), this also causes large delays in the PTG. We are not concerned with fairness, but rather, overall program completion time, where preemption might make sense to use in some cases but not in others.

Our framework consists of two major steps: *task allocation* (optional) and *task scheduling*. While some algorithms include task allocation as part of their task scheduling step, other algorithms can significantly benefit from performing task allocation as a separate step before doing task scheduling. Once a task is allocated to a set of processors (CPU and/or GPU), the task is executed on the assigned set of processors for the entire duration of the

stream program. Different tasks processing different inputs are scheduled using our task scheduling algorithm.

5.3 Problem Formulation of Task Allocation of Stream Programs

We use the following assumptions for our problem formulation and proofs. We use notations in Table 5.1 to describe a set of tasks, processor specification, and other important information for the problem formulation.

Assumptions: (1) at least one task has a dependency on itself from previous iteration – this is suitable for applications where decision is made based on the order of processing; (2) cost per task is static as assumed in Satish's; (3) a task (once its profile is chosen) runs on that same set of processors using the same profile in subsequent iterations; (4) preemption is available. We focus on scenario where processing time greatly exceeds communication cost.

Table 5.1: Notations for the task allocation problem

Notation	*Description*
$T = \{T_1, T_2, \ldots, T_n\}$	*Set of n indivisible tasks in a stream program*
$D = \{D_1, D_2, \ldots, D_d\}$,	*Set of d computing devices (processors) in the system where D_k denotes device (processor) k*
$P = \{P_{1,1}, P_{1,2}, \ldots, P_{n,l}\}$	*Set of profiles where P_{ij} denotes profile j for task i where j is an integer corresponding to a processor configuration B*
$B(b_1, b_2, \ldots, b_d)$	*A processor configuration in which b_m is 0 if processor m is not used, 1 if it is used, and 2 if it is used only for a comparatively small amount of time as a support processor (such as a CPU used to launch GPU code)*
$P^T_{i,j,k}$	*Time spent on task i by processor k using profile $j = \Sigma_i(b_i \cdot 3^{i-1})$; the profile is the number represented by the trinary sequence B.*
$P^M_{i,j,k}$	*Memory required by task i on processor k using profile j*
$S_{i,j}$	*$S_{i,j}$ = 1 if profile j is used for task i;* *$S_{i,j}$ = 0 otherwise*
C	*Set of constraints; for instance, C^M_k denotes the amount of memory available for processor k*

For the problems where tasks do not have dependency between iterations, steady state scheduling algorithms may be considered instead [72].

Since a task may have different portions of itself running on different processors, we use profiles of different processor configurations to model this requirement. Suppose $D = \{D_1, D_2, D_3, D_4\}$ where D_1 and D_2 represent CPU_1 and CPU_2, respectively and D_3 and D_4 represent GPU_1 and GPU_2, respectively. For example, the processor configuration *B(1,0,1,0)* indicates that only CPU_1 and GPU_1 are utilized. For this configuration, the profile number is $1 \cdot 3^3 + 0 \cdot 3^2 + 1 \cdot 3^1 + 0 \cdot 3^0 = 30$. $P^T_{i,j,k}$ is used to capture execution times taken on task *i*, profile *j*, and by processor *k*. For the above profile number 30, we have values for $P^T_{1,30,1}$ and $P^T_{1,30,3}$. For instance, $P^T_{1,30,1}$ of 3 indicates that processor 1 (CPU_1) spends 3 time units to execute task 1 whereas $P^T_{1,30,3}$ of 6 indicates that processor 3 (GPU_1) takes 6 time units on task 1. Invalid profiles have special values to indicate that they are not valid. For instance, configuration *B(0,0,1,1)* is not valid since we cannot run GPU kernel without CPU (even a tiny amount of CPU time is usually spent to setup and launch a GPU kernel). While CPU might only be used for support (e.g., to launch a GPU kernel), it may instead be taken full advantage of in order to do additional processing. The trinary nature of B handles this difference.

Problem Definition: Given *T* by a program specification, *D* and *C* by a hardware specification, and *P* through automated profiling on the intended hardware, *find the matrix S that minimizes makespan approximated by the maximum load (execution time) across all processors as shown in Objective (5.1).*

$$\underset{S}{\text{minimize}}\left(\text{maxload} = \max_k \left[\sum_{i,j} \left(P^T{}_{i,j,k} \cdot S_{i,j} \right) \right] \right) \tag{5.1}$$

$$\forall i : \left[\sum_j \left(S_{i,j} \right) = 1 \right] \tag{5.2}$$

$$\forall k : \left[\sum_{i,j} \left(P^M{}_{i,j,k} \cdot S_{i,j} \right) \leq C^M{}_k \right] \tag{5.3}$$

In Objective (5.1), the time spent on task *i* by processor *k* using profile *j*, $P^{T}_{i,j,k}$, is accumulated with other tasks on the same processor if $S_{i,j}$ is 1 (i.e., profile *j* is used for task *i*). Constraint (5.2) allows only one profile per task. Constraint (5.3) describes a memory constraint that the amount of memory required by all programs allocated to a particular processor cannot exceed that processor's associated memory capacity. Some additional constraint equations could easily be added using this model to support real-time deadlines or other desired effects.

The formulated problem is an integer linear programming problem, which is NP-Complete. Before we discuss our heuristic solution, we present Theorem 1 used to justify the validity of Objective (5.1).

Theorem 1: Minimizing makespan of a stream program over a large number of inputs is approximately equivalent to minimizing the maximum load (execution time) across all processors.

Makespan of a stream program is the length of the critical path of the program PTG over all its inputs. In this paper, we focus on the set of problems where communication cost between processors is considered negligible compared to the computation cost. GPU is a good candidate for this problem set. We set the edge cost to zero and include in a node's cost any computation cost involved to transfer data between tasks. Any communication cost incurred outside of computation cost could be accurately represented by creating a virtual node on a virtual processor that represents the data transfer cost of that communication (edge), but incurs no additional computation cost. We will investigate this possibility as part of our future work.

With Assumptions (2) and (3), the execution time in different iterations is same. Our proof does not need to consider statistically iid. Assumption (3) simplifies the problem and avoids processor switching overhead.

Table 5.2: Notations for proving Theorem 1

Notation	*Description*
$N_{i,j}$	*Node represents task j of iteration i for input i (e.g., frame i)*
g_i	*A graph containing nodes and edges of the original PTG for only iteration i; all g_i's nodes and edges are same between different i's*
G_i	*PTG consisting of all nodes from iteration 1 to iteration i including added edges between iterations*
$cost_N(N_{i,j})$	*Node cost – execution time of node $N_{i,j}$*
$cost_P(N_{i,j} \rightarrow N_{i,k})$	*Path cost – sum of the time taken by all nodes along the critical path from $N_{i,j}$ to $N_{i,k}$, including $N_{i,j}$ and $N_{i,k}$*
$N_{i,c}$	*A cut point node at task c in iteration i*

Table 5.2 shows the notations used for the proof of Thereom 1. Figure 5.2 shows an unrolled PTG of Figure 5.1 over a number of iterations, each processing one input. Note that an unrolled graph is used only for the proof; with the proof, it is no longer necessary to unroll the PTG for our heuristic solutions.

We first consider simple cases for homogeneous systems. We present the proof for case I when the number of tasks is at most the number of processors. Each task is assigned a distinct processor for maximum parallelization. We then map the problem with the number of tasks more than the number of processors (case II) to that of case I and utilize the results of the case I proof. Next, we handle choice of heterogeneous profiles. We use notations in Tables 5.1 and 5.2 for the proof.

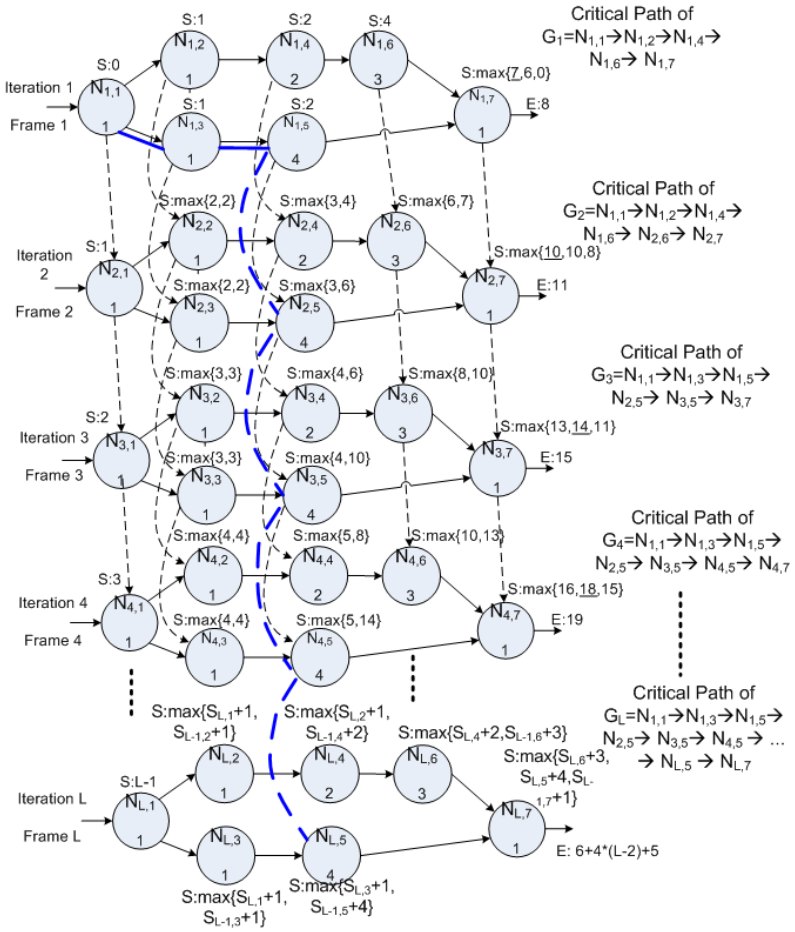

Figure 5.2: Unrolled PTG of Figure 5.1. Notations are the same as those in Figure 5.1 and Table 5.2, where S represents the start time of a node. The *i*th iteration processes frame *i*. A black dashed line between iterations indicates that a task in the *i*th iteration must finish first before the same task can proceed in the *i+1*th iteration. For the unrolled graph of two iterations G_2, the cut point node is $N_{1,6}$. For unrolled graphs of three iterations or more, the cut point node becomes $N_{1,5}$ and remains at this node. The critical path of the graph of frame 1 to frame *L* is labeled as a blue dashed line.

Case I: The number of tasks is at most the number of processors

Base step: G_1 consists of nodes $N_{1,1}$, …, $N_{1,n}$ with the same number of nodes and edges as in the original PTG of the given stream program. Let $N_{1,1}$ denote the source node and $N_{1,n}$ denotes the sink node (virtual nodes with no cost can be created for the sake of having a single source and single sink node if necessary). The makespan of G_1 is same as that

of the original PTG. The critical path is the path from the source node to the sink node with the longest length. See the critical path of G_1 in the upper-right corner of Figure 5.2.

We add nodes of the next iteration g_2 to G_1 to form G_2. Each node in g_1 will have a directed edge drawn to its corresponding node in g_2 to represent a temporal dependency (as in Satish's technique [22]). Thus, for all task j, a directed edge is added from $N_{1,j}$ to $N_{2,j}$. We already know the critical path of G_1; we compute a new critical path for G_2. Although the source node of G_2 is the same as that of G_1, the sink node changes from the final node of iteration 1 ($N_{1,n}$) to the final node of iteration 2 ($N_{2,n}$). The only way to draw a path from a node in iteration 1 to a node in iteration 2 is using one of the newly added edges that goes from $N_{1,j}$ to $N_{2,j}$ for some j (Assumption 1). *It would not be possible to form a new critical path to $N_{2,j}$ without going through one of these edges.* The new critical path of G_2 will include a "cut" at some point j in $N_{1,j}$ to $N_{2,j}$, with a new edge from $N_{1,j}$ to $N_{2,j}$. *The new critical path will run as usual along nodes in the first iteration until its cut point, $N_{1,j}$ where it then jumps to $N_{2,j}$ and continues along that path*. For instance, in Figure 5.2, the critical path of G_2 results in the makespan of 11 which is derived from the earliest time its input is ready (the maximum end time of its immediate preceding node in the same iterations and the preceding node in the previous iteration or $\max\{N_{2,6}, N_{2,5}, N_{1,7}\}$ which is $\max\{10, 10, 8\}$. To break the tie for the same cost, we choose the first cost. Hence, to get all nodes in the critical path, we trace backward from $N_{2,6}$ using a similar method until the source node $N_{1,1}$ is reached. The critical path of G_2 goes from $N_{1,1}$ to $N_{1,6}$, cuts at $N_{1,6}$ to $N_{2,6}$, and continues from $N_{2,6}$ to $N_{2,7}$.

Let $N_{1,c}$ be the current cut point. The critical path length of G_2 or the makespan considering only 2 iterations can be derived using Expression (5.4) where $i = 2$.

$$cost_P(N_{1,1} \rightarrow N_{1,c}) + (i-2)*cost_N(N_{1,c}) + cost_P(N_{i,c} \rightarrow N_{i,n}) \qquad (5.4)$$

Inductive Step: We consider the critical path in the base case and a new candidate critical path created in a similar way to form G_2. Let $N_{1,nc}$ represent a cut point of a new iteration $i>2$. $N_{1,nc}$ is different from $N_{1,c}$ when there exists node $N_{1,nc}$ different from $N_{1,c}$ that

satisfies Inequality (5.5). In other words, a new critical path cutting through a new cut point results in a higher execution time.

$$cost_P(N_{1,1} \rightarrow N_{1,c}) + (i\text{-}2)*cost_N(N_{1,c}) + cost_P(N_{i,c} \rightarrow N_{i,n}) <$$
$$cost_P(N_{1,1} \rightarrow N_{1,nc}) + (i\text{-}2)*cost_N(N_{1,nc}) + cost_P(N_{i,nc} \rightarrow N_{i,n}) \qquad (5.5)$$

As the number of iterations increases, the critical path will divert to use the maximum node cost as the cut point since the length of the critical path of each G_i increases the most when $cost_N(N_{1,nc})$ is the highest. Figure 5.2 shows that in iteration 3, the cut point changes to $N_{1,5}$ instead of $N_{1,6}$.

Over a large number of inputs (thousands or millions of inputs), the iteration term $(i\text{-}2)*cost_N(N_{1,c})$ dominates the other two critical path terms. This is because the maximum value of the critical path terms combined is approximately limited by, at most, the sum of the costs of all the nodes in the original PTG (as in the case of a serial program where the critical path runs through every task). The iteration term will eventually converge to using the node with the highest cost. When i is equal to the number of nodes plus three, the iteration term is still at least the sum of the critical path terms, in the worst case. As i increases, the iteration term continues to grow while the critical path terms remain the same. The significance of the critical path terms, in the worst case, may be approximated by the number of nodes divided by the number of iterations. Because we expect the number of iterations i to exceed the number of nodes of the original PTG by a significant amount – perhaps 10-100 nodes in the original PTG compared to over a million iterations for the overall program – the significance of the critical path terms could be lower than 100 ÷ 1 million = 0.01%. Hence, minimizing makespan is approximately equivalent to minimizing the maximum cost node.

Case II: The number of tasks is more than the number of processors

The primary difference when there are fewer processors than tasks is that some tasks must share a processor with another task. This creates competition or contention for a processor's compute time. Instead of all tasks running in parallel, tasks that use the same

processor need to run in serial of each other, but tasks that run on different processors can still run in parallel.

To handle this difference, we need to modify Expression (5.4). While the meaning of the expression remains the same, we modify the per-iteration cost, which originally used the cost of a single node (representing a single task). Instead, the per-iteration cost uses the maximum processor load among the processors on which that task executes. Because the load of a processor is the sum of the costs of all tasks that run on that processor, it is easy to see that the effective time between a task being executed on successive iterations is the total load of the processor that task runs on. Thus, it is correct to use the processor load time instead of the node's cost, as seen in Expression (5.6). We use the *maxload* as defined in Objective (5.1). For tasks that utilize multiple processors, we use the maximum load out of all the processors the task uses, as this will determine the time between successive iterations of the same task. While the meaning of the costs representing the critical paths to and from the cut point remain the same from Expression (5.4), these critical paths must reflect the critical path taking processor contention into account. These terms are still insignificant compared to the iteration term from Expression (5.6).

$$cost_P(N_{1,1} \rightarrow N_{1,c}) + (i\text{-}2) * maxload + cost_P(N_{i,c} \rightarrow N_{i,n}) \qquad (5.6)$$

Dealing with heterogeneous choice among tasks: When we have heterogeneous processors, the same task is likely to take different times on different processors, resulting in different costs for the same task. While some have proposed a normalization of all costs to a single reference processor speed [25], this approach does not work in practice due to variable speedups dependent on the task implementation and processor types (e.g., when running on GPU instead of only CPU, one task might get 4x speedup and another task might only get 2x speedup). To properly minimize the overall cost, we choose the profile (and thus allocation) that will minimize this maximum load across all processors.

5.4 Load Balancing Algorithms for Task Allocation

5.4.1 Brute Force with Pruning

We implemented a simple brute force solver for load balancing. Because our method does not require loop unrolling, the number of tasks can remain fairly small for many programs. The brute force algorithm tries every combination of load balancing of tasks (based on their profiles); however, it prunes duplicates to reduce redundancy (e.g., having two homogeneous processors' loads exactly swapped) and also keeps a running best solution to prune searches when any load already exceeds the best solution. With our generated PTG benchmarks described in the performance evaluation, we can compute an exact solution for 20 tasks on 2 CPUs and 2 GPUs in under one second. Even though 20 tasks may be completely realistic in many cases, brute force does not scale well. Therefore, we introduce a simple greedy algorithm called *K-HIT*.

5.4.2 K-HIT Greedy Algorithm

K-HIT considers all combinations of profiles using K tasks at once out of the total number of tasks in the PTG and selects the one combination with the minimum makespan. This is applied repeatedly until no further improvement can be made. This is applied to load balancing for a single iteration. This is a greedy solution and will eventually result in what is probably a local minimum rather than a global minimum. Increasing K allows the algorithm to break local minima more easily; however, it also increases the exponent of this polynomial time algorithm $O(m^K)$ where m is the maximum number of profiles for a task. We found that K=3 works well for most of our simulated graphs described in Section 5.5 and that K=4 does not improve the quality of the solution by much, if at all. Without HIT, this problem is similar to the n-partition problem, a well-known NP-Complete problem [22], for which several heuristic algorithms exist. There are also other heuristic algorithms specifically targeted toward load-balancing. It may be possible for such algorithms to be augmented to support HIT and used in place of K-HIT.

5.5 Algorithm for Task Scheduling

We schedule all tasks for one iteration at a time, starting from the first iteration until the last iteration when the last input is processed. Within a single iteration, we use EFT to schedule all tasks for that iteration. We call this variation on EFT for stream programs Stream-EFT. Although scheduling is performed for each iteration, it is done independently of other iterations; thus, unrolling the task graph is not necessary, saving time and memory. However, some state information is saved between iterations. Bubble filling is able to reduce idle time between iterations compared to methods that only repeat the same schedule of one iteration for all the iterations.

5.6 Experimental Setup and Results

In order to compare our proposed algorithm with related work, we generate various synthetic stream parallel task graphs as input. The goal is to determine how well each scheduling algorithm works with different types of stream programs. We generated our own PTGs for benchmarking.

5.6.1 Graph Generation

Problems with Existing Benchmarks

Although some existing PTG data sets exist for benchmarking purposes (e.g., [73]), they are not necessarily intended for stream programs. We also did not find any that supported HIT. Because we wanted better control over the parameters of the graph (such as the degree of parallelism, number of tasks that support HIT, and to what varying degree they support HIT), we implemented our own graph generator and simulator for testing.

Types of Graphs and Graph Generation

A common target stream program used for benchmarking is that of MPEG compression. The program usually involves a long pipeline of steps that includes some small amount of parallelism and, overall, many serial tasks [22]. We wanted to generate similar graphs as seen in real-world scenarios. Thus, with our graph generator, we can specify the minimum, average, and maximum degree of dependencies between each task; this

determines the amount of parallelism in the overall graph. We also wanted to compare algorithms against mostly serial and mostly parallel task graphs to see the effects that those have on performance. We primarily focus on the hybrid PTG, which contains a realistic amount of both serial and parallel computation. We simulated and averaged the results for the simulations of 20 hybrid PTGs.

For the serial graph, we chain together 20 nodes in series so that each node has only one dependent, except for the final sink node. For the parallel graph, we start with one source node, link 18 dependent nodes directly from the source node, then link all those nodes into a final sink node. So, those 18 node can all run in parallel.

Table 5.3: Descriptions of scheduling algorithms

Algorithm	*Description*
Cyclic	*Cyclic scheduling with HIT added; brute-force is used to compute the optimal cyclic schedule for one cycle, which is repeated.*
PU w/o Pre. (10)	*Partial unrolling (PU) without preemption; PTG unrolled 10 iterations for computing a schedule with DLS augmented to support heterogeneous choice, which is then repeated as necessary to fill all required iterations (e.g., 2^{20} iterations).*
PU w/ Pre. (10)	*Same as previous, but with preemption enabled.*
K-HIT + S-EFT w/o Pre.	*Proposed K-HIT and Stream-EFT algorithm without preemption; load balancing is computed over a single iteration, then the resulting task-processor allocation is applied to the original PTG for every iteration.*
K-HIT + S-EFT w/ Pre.	*Same as previous, but with preemption enabled.*
K-HIT + S-EFT w/o Pre. (10)	*Proposed work limited to scheduling for only 10 iterations, without preemption; load balancing is computed over a single iteration with K-HIT, then the resulting task-processor allocation is applied to the original PTG for 10 iterations for computing a schedule with Stream-EFT. The resulting schedule is replicated end-to-end to account for all required iterations.*
K-HIT + S-EFT w/ Pre. (10)	*Same as previous, but with preemption enabled.*
K-HIT + S-EFT w/ Pre. (1000)	*Same as previous, with preemption, but scheduling on the original PTG for 1000 iterations.*

For generating the randomized hybrid PTGs, we chose an average of 20 tasks for the PTG, 1-3 dependents for each task, execution time for each task on the CPU from 1-1000 time units, an average of 40% of those tasks as being HIT-enabled, and a realistic speedup range for HIT-enabled tasks on GPU from 0-9 times speedup, where zero represents the same cost (no speed gain) from heterogeneous processor (GPU) usage. The ranges for these parameters were chosen in an attempt to get benchmarks to be as realistic as possible with real-world programs and GPU speedups.

5.7 Task Scheduling Algorithms and Features

The variations of task scheduling algorithms are listed in Table 5.3. Some minor variations have been included to show the impact that those variations have on various algorithms.

5.8 Results

The results of each scheduling algorithm and their variations are shown in Tables 5.4 through 5.6. The makespan as well as the time taken to generate a schedule and memory usage of the scheduling algorithm used to achieve that makespan are included. All programs were scheduled for 2^{20} iterations (equivalent to about 10 hours worth of video frames at 30 frames per second). Results are shown for a purely serial task graph, purely parallel task graph, and the average of 20 randomly generated hybrid task graphs, which include some degree of serial and parallelism between tasks. For the makespan, a percent relative to the baseline algorithm "K-HIT + S-EFT w/ Pre." is used. For example, a value of 200% indicates that the algorithm takes twice as long. For all three tables, a lower value is more desirable. All of these were computed on a quad-core 1.73GHz i7 (x64) computer with 8GB of RAM. The optimal cyclic schedule could not be computed with brute force for many parallel task graphs because brute force examines too many permutations (over one day of computation time was spent without any reasonable result). We also simulated DLS with preemption on a fully unrolled graph, and it did not complete within one day of computation time.

Table 5.4: Makespan

Algorithm	*Serial*	*Hybrid (20)*	*Parallel*
Cyclic	*186.92%*	*126.08% **	-
PU w/o Pre. (10)	*153.53%*	*137.10%*	*116.52%*
PU w/ Pre. (10)	*151.27%*	*135.06%*	*112.53%*
K-HIT + S-EFT w/o Pre.	*368.22%*	*231.63%*	*100.00%*
K-HIT + S-EFT w/ Pre.	*100.00%*	*100.00%*	*100.00%*
K-HIT + S-EFT w/o Pre. (10)	*368.23%*	*226.26%*	*100.10%*
K-HIT + S-EFT w/ Pre. (10)	*139.13%*	*115.04%*	*100.10%*
K-HIT + S-EFT w/ Pre. (1000)	*100.46%*	*100.20%*	*100.04%*

Table 5.5: Time Taken to Generate a Schedule (ms)

Algorithm	*Serial*	*Hybrid (20)*	*Parallel*
Cyclic	*47*	*1879452* [*]	*> 1 day*
PU w/o Pre. (10)	*26*	*18*	*42*
PU w/ Pre. (10)	*42*	*33*	*41*
K-HIT + S-EFT w/o Pre.	*13961*	*22853*	*39590*
K-HIT + S-EFT w/ Pre.	*32542*	*48890*	*116561*
K-HIT + S-EFT w/o Pre. (10)	*60*	*42*	*66*
K-HIT + S-EFT w/ Pre. (10)	*64*	*64*	*71*
K-HIT + S-EFT w/ Pre. (1000)	*92*	*126*	*173*

[*] Due to the enormous computation time required, the result for only the cyclic schedule is shown for one hybrid graph rather than an average of the 20.

Table 5.6: Memory usage (MB)

Algorithm	*Serial*	*Hybrid (20)*	*Parallel*
Cyclic	*1*	*1* *	-
PU w/o Pre. (10)	*1*	*1*	*1*
PU w/ Pre. (10)	*1*	*1*	*1*
K-HIT + S-EFT w/o Pre.	*1*	*1*	*1*
K-HIT + S-EFT w/ Pre.	*852*	*819*	*640*
K-HIT + S-EFT w/o Pre. (10)	*1*	*1*	*1*
K-HIT + S-EFT w/ Pre. (10)	*1*	*1*	*1*
K-HIT + S-EFT w/ Pre. (1000)	*1*	*1*	*1*

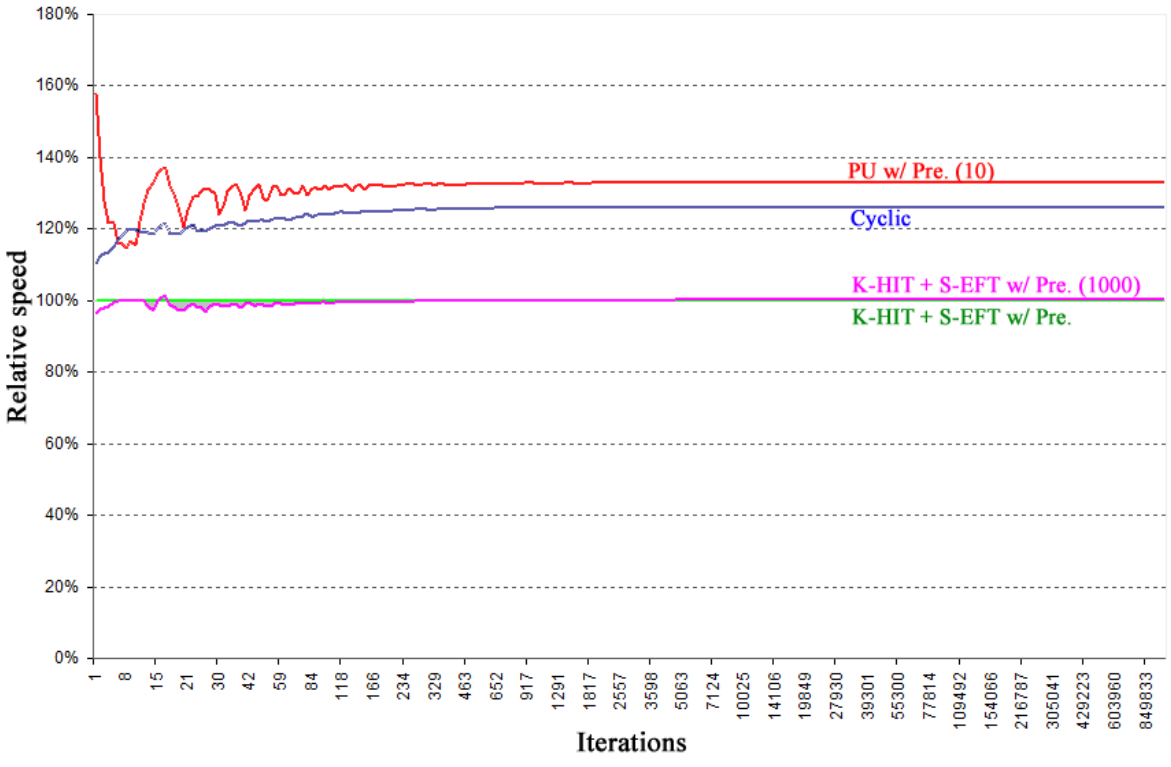

Figure 5.3: Makespan of select algorithms with respect to the baseline for variable number of iterations for a hybrid PTG. Although some variability exists at lower numbers of iterations, as the number of iterations becomes very large, the relative makespan stabilizes to show a clear pattern.

5.9 Discussion

The proposed work, K-HIT for load balancing and Stream-EFT for scheduling, with preemption, yields a theoretical near-optimal result that can be compared against. For example, this algorithm yields a final result for 2^{20} iterations that is, on average, only 0.00026% larger than the maximum processor load (from the load balancing step) multiplied by the 2^{20} iterations. If the load balance chosen is optimal, then this scheduling algorithm also yields an optimal schedule to within a negligible amount of deviation.

In Figure 5.3, several algorithms are shown, simulated with varying numbers of iterations from 1 to 2^{20}. Although at lower numbers of iterations, results are too variable to draw any conclusions, when higher numbers of iterations are simulated (as would be seen in stream programs), results become consistent. Additionally, for our baseline algorithm, as the number of iterations increases, the makespan asymptotically approaches the load balance multiplied by the number of iterations, as shown in Section 5.3.

Thus, computing an optimal load balance is very important. For small task graphs (e.g., 20 tasks), regardless of the amount of serial or parallelism between tasks, the actual optimal load balance is fast to determine with brute force; thus, an overall optimal schedule can be found in a short amount of time. For larger task graphs where brute-force is unrealistic, the heuristic algorithm used to compute the load balance is the most influential factor to the final makespan, as the final makespan can still be computed to within a negligible amount of deviation from the load balance result multiplied by the number of iterations.

K-HIT is useful even by itself, when applied to existing task scheduling algorithms. It can be used to supplement or as an alternative to the task allocation step of task scheduling algorithms. For existing scheduling algorithms that were augmented with heterogeneous choice, this is an especially important step, as these existing algorithms do not function well with the additional nondeterminism of having to choose between different implementations of the same tasks. K-HIT makes that choice in the task allocation step, allowing existing algorithms to work in the same way that they were originally designed to work.

For Stream-EFT, we schedule using the original PTG that is not unrolled; however, to allow preemption, we consider free time created from bubbles in prior iterations. Keeping track of these bubbles over millions of iterations requires high (but manageable) bookkeeping costs of time and memory. This cost may be somewhat reduced by computing a schedule on-the-fly during execution time since prior bubbles of time that have already elapsed no longer need to be considered. Most of the time and memory is spent traversing and maintaining a complex tree and linked list structure that efficiently keeps track of the bubbles. Near the end of a program's execution, it is unlikely that the bookkeeping for bubbles at the beginning of the program are useful. So, these resources could be freed at the potential cost of a less efficient makespan.

Preemption is very important for Stream-EFT to get good results. Thus, if preemption is not desired, while K-HIT is still useful, it should be combined with an alternate stream scheduler.

These algorithms establish a baseline proof-of-concept of our approach. The combination of using the proposed K-HIT and Stream-EFT algorithms along with an idea similar to the partial unrolling work where fewer iterations are scheduled, but with many more iterations "unrolled" (1000) than originally proposed (10) by [22], yields nearly as good a result without the disadvantages of high computation time and memory requirements as in the proposed work alone. Note that the PTG is never actually unrolled in this combination since K-HIT and Stream-EFT do not require unrolling – only the number of iterations scheduled is different. Because there are fewer bubbles created in the process of scheduling over only 1000 iterations, the memory usage never exceeded 1MB for the graphs tested. It is then relatively fast to replicate the resulting schedule of 1000 iterations to account for all 2^{20} iterations. Due to its very low memory and computation cost, this approach may be the preferred method in practice.

5.10 User-Mode Task Scheduler

In order to make our theoretical task scheduler practical, we investigated options to implement a task scheduler without modifying the operating system kernel. Because operating systems generally implement their task scheduler in kernel-mode, it is not possible

to easily specify our own scheduling algorithm for an existing operating system. So, we use an alternate method where we approximate a task scheduler completely in user-mode.

In an operating system, the task scheduler is responsible for choosing a thread or process (depending on scheduling granularity) to run on each processor at any given time. Although the actual algorithm for task scheduling may vary, the mechanisms for multitasking – actually getting tasks to switch – are typically the same. There are two major types of multitasking: cooperative and preemptive. In cooperative multitasking, a task runs for as long as it wants, until it decides to yield its processing time to another task. This means that one task can completely block a processor from being utilized by any other task if it never yields. In many situations, this is undesireable. With preemptive multitasking, a task is given a set amount of time to run, after which it may be forcefully swapped out for another task by the operating system. This is usually made possible through some protected processor functionality only available to the operating system.

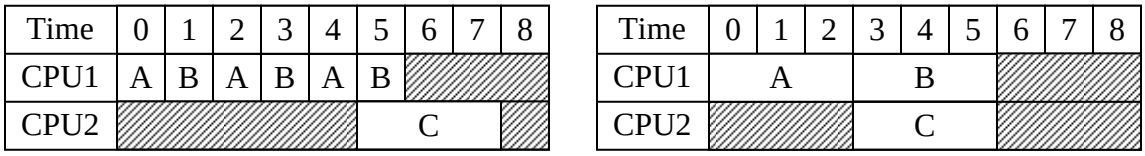

Figure 5.4: An example of (a) preemptive round-robin multitasking, left, and (b) cooperative multitasking, right.

Although preemptive multitasking is generally used to give some fairness to all threads by forcefully cycling through them in some fashion, it can also prove to be less optimal for a given program. This can be shown with a simple example in Figure 5.4. Consider a two processor system with three tasks, A, B, and C, each taking 3 time units to finish. As a restriction, tasks A and B can only run on the first processor (e.g., some processor-specific code in a heterogeneous system), while task C can only be run on the second processor. Task C depends on task A finishing. In the case of preemptive multitasking, suppose that each task is allowed to run for one time unit before it is forcefully swapped out for another task, and that tasks are switched in round robin fashion. In this situation, shown in Figure 5.4(a), the first processor would run A for one time unit, then B

for one time unit, then A, B, A (where it finishes at time 5), and finally B (where it finishes at time 6). Task C can finally start at time 5, where it takes 3 time units, so the entire program finishes at time 8. If, instead, we were to use a cooperative task scheduler, shown in Figure 5.4(b), we could have run A first continuously for 3 time units where it would finish at time 3. Then, task B can run on the first processor starting at time 3 while task C can run on the second processor also starting at time 3. Both tasks finish at time 6, so the overall program would finish at time 6. Operating system schedulers generally do not know anything about the underlying thread dependencies of a program, so it is possible that a preemptive scheduler (which is what typical operating systems like Windows and Linux use) will result in a less optimal program execution.

For our theoretical static task scheduler, we can construct a complete schedule based on the execution times of each task. In the simple case where one task finished and another starts immediately after, it is easy to see that cooperative multitasking directly applies. In the case where we have scheduled a task to fill in a bubble that is shorter than the total duration of a task (and thus it must be broken into more than one part), we may need to use preemptive multitasking to forcefully halt a task at a given moment in time, then start the other task. Thus, to implement our user-mode task scheduler, we will utilize both cooperative and preemptive strategies.

In Windows, there are several key functions used to make this possible (Linux and some other operating systems have similar functions) – CreateThread to create new threads corresponding to tasks, SetThreadAffinityMask to lock individual threads to specific processors, SuspendThread to pause threads, ResumeThread to resume threads, and Sleep to yield a task's currently scheduled timeslice.

5.10.1 Design and Implementation

Although there are several ways to implement a user-mode scheduler, we implement it as follows. We begin by creating all task threads initially suspended from a supervisor thread. The supervisor thread represents the functionality of an operating system task scheduler and is therefore responsible for suspending and resuming threads. Although we do not have access to the true hardware preemption resources that the operating system does, we

can emulate this to some degree. We use only a single supervisor thread to control all processors. The supervisor thread uses some task scheduling algorithm to choose a thread for each processor. In order to execute a task for a specified amount of time on a given processor, it first sets the thread's processor affinity mask to that single processor, then resumes that thread (which was created in a suspended state). This is done for each processor in the system.

In this way, as long as there are no outside factors, each processor is running the specified task we gave to it and nothing else. All other threads of our program have been suspended, so they are not considered for scheduling by the operating system scheduler. Only the active threads are scheduled, and they all have unique processor affinities, so they will not interfere wih each other. In reality, though, there will be outside factors (other threads running in the system from other programs), there is little we can do to guarantee complete non-interference for our user-mode task scheduler. However, we can attempt to reduce this interference by increasing thread priority (making it less likely that other threads will be swapped onto our specified processor) and making sure not to run other programs while we run our user-mode task scheduled program. So, with some uncertainty, the operating system closely obeys our task scheduler.

When a thread has finished processing its data, it calls Sleep to yield its timeslice in a cooperative multitasking way, so that a new thread can be scheduled. Our user-mode task scheduler (now running as the task's thread as opposed to the supervisor thread) intercepts this call to Sleep in order to choose a new task to run for the processor that task was running on. The old thread must be stopped with SuspendThread while a new thread to be scheduled will be started back up with ResumeThread. A new task is chosen, and the appropriate functions are called to perform these operations, setting the same processor affinity for the new task as the old task (to reuse that specific processor). SuspendThread is called last in the sequence of events, since the scheduler is being run from the same thread that will become suspended. If the thread suspends itself before resuming a new thread, then the scheduler will have effectively stopped itself, and the other thread would not have been resumed. We can repeat this scheduling process until all tasks have completely finished and their threads

exited. When all worker threads have exited, the user-mode task scheduler determines that the program has effectively finished.

While the supervisor thread is not necessary to support cooperative multitasking, we utilize it to support preemptive multitasking. Because the underlying operating system works with a preemptive scheduler, it is possible to schedule a non-suspended thread (the supervisor thread) to wake up based on a timer, thereby potentially preempting a running task, with some margin of error that is based on the underlying operating system's preemptive scheduler. We schedule the supervisor thread to wake up based on the time of the earliest expected task that should be preempted. When our supervisor thread wakes up, it verifies that the expected thread is still running, suspends it, then resumes a different thread on that same processor that completes the schedule. The supervisor thread again determines the next earliest time it should wake up and goes back to sleep based on that timer.

5.10.2 Dynamic Scheduling

The method described is sufficient to implement a task scheduler in user-mode on top of a preemptive multitasking operating system. However, because there are outside factors, and the granularity of our preemption is based on the granularity of the underlying operating system (potentially, the length of a timeslice), there are some variable factors that our original statically generated schedule did not and could not necessarily account for. Additionally, for a real program, the time a specific task takes to execute for an iteration may not necessarily remain static, but instead, follow some distribution depending on its input data, the time a task takes may vary by a non-trivial amount.

We can somewhat account for this with a static scheduler by using the worst case time of a task for every iteration (so that the static schedule remains static), but then we waste processor time when those tasks finish before their worst case time. We could also use the average case time (or some other composited time based on each task's profile), but then the static schedule would not be able to be followed exactly.

Instead of following a precomputed static schedule exactly, we can apply a set of rules to determine which tasks should be scheduled at a given point in time based on what tasks have already finished which iterations. Although this does not follow exactly all the

conditions of the static scheduler proposed, it is still a close approximation, and would work in a practical implementation. Currently, the static task scheduler attempts to schedule each iteration one at a time, using EFT within an iteration. Thus, it follows that a dynamic scheduler should prioritize tasks based on iteration number first, and EFT second. Because dependencies might not always be immediately satisfied, some future iterations of some tasks may be able to run before an earlier iteration of a different task. However, we still want to enforce the priority of iteration number. So, we need to potentially preempt some task (which has filled a bubble) with another task. The design of our user-mode scheduler allows for this, as long as we know when to preempt.

At any given point in time, we know the latest iteration that each task has processed, and we also have the dependencies between tasks. So, we know which tasks and which iterations are available to be scheduled – call this set S. Its initial state consists of the first iterations of any tasks that do not depend on any other tasks (i.e., purely producer tasks with no inputs). While we continue to lock tasks to processors, we keep only a single set S for the whole system. At some point in time, then, we can properly schedule based upon our priority of iteration number, and then EFT next, from our set S. This scheduling would not change until our set S changes. This only occurs when another task in the system has finished, potentially satisfying some dependencies. At this point, the finished task is removed from S and all of the fully satisfied dependent tasks can be inserted into S. For each of these new tasks, if their priority (based on iteration number and then EFT) for their specified processor is higher than a currently executing task on that processor, then we preempt the currently running task for the new task. In this way, we follow the philosophy of the statically scheduled Stream-EFT, but we never rely on scheduling for specific points in time that our static schedule relied on. We can schedule entirely dynamically based on the changes in satisfied dependencies.

In the case where all tasks have static execution times across all iterations (as in the static scheduler), this method for dynamic scheduling would result in the same schedule being generated. However, because a dynamic scheduler also works for dynamic execution times, and we expect execution time to be dynamic in practice, it may be more realistic to implement a dynamic user-mode scheduler than a static user-mode scheduler.

For a proof of concept, we implemented a round-robin dynamic user-mode scheduler in SAPPHIRE with promising results. Just by using the user-mode scheduler, we saw a reduction in the execution time of some sample SAPPHIRE benchmarks by about 20%. This speedup compared to letting the OS do all scheduling is attributed to several factors, including longer virtual timeslices, less contention of threads for processors (i.e., threads can finish processing on their own before the OS interrupts their execution with another thread), and the ability to lock out certain processor cores (for example, we found that disabling hyperthreading by disabling scheduling on certain virtual CPU cores yielded a non-trivial speedup for some types of tasks for which hyperthreading provided little to no benefit, but increased the time to finish of both tasks executing).

Although we implemented a round-robin scheduler, due to some design limitations in SAPPHIRE's work loop, it would have been a significant undertaking to implement the user-mode scheduler for Stream-EFT in our middleware. SAPPHIRE's work loop automatically queues and processes all available input data at some given point in time (as an optimization), whereas Stream-EFT needs more fine grained control over which pieces of data are processed. For example, if ten VIDEO packets are available to process, all ten will be processed before the work loop yields execution. Stream-EFT would demand that we potentially reschedule tasks in the system after reconsidering what new tasks were satisfied after completing each iteration of a VIDEO packet (as well as dependencies satisfied by other tasks in the system). From simulations of our scheduling algorithm, without this fine grained control, we saw an increase in execution time. Therefore, without redesigning our work loop, a user-mode scheduler for Stream-EFT would not be effective.

5.11 Summary and Future Work

In this chapter, we present a theroetical framework for static sheduling for stream programs considering execution profiles of each task on a heterogeneous system. Our proposed K-HIT and Stream-EFT method achieves near optimality for high numbers of iterations and is comparable to or better than existing work in many respects. We also proposed a faster approach that further reduces memory and computation cost, while still achieving a high quality schedule. For future work, we plan to extend our method to work

with statistical profiles of tasks as opposed to a single static cost and to implement the discussed bandwidth virtualization to support memory-intensive style tasks. As a proof of concept, we show that implementation of a user-mode scheduler is possible for round-robin scheduling, but a significant change in the design is needed to fully support Stream-EFT scheduling.

CHAPTER 6. EVALUATION OF SAPPHIRE FOR DEVELOPMENT OF MEDICAL VIDEO ANALYSIS APPLICATIONS

Two medical video analysis applications were developed using SAPPHIRE. We developed the first application called "EM-Capture" for automatic detection of endoscopic videos, which takes a stream of images and records those corresponding to an endoscopic procedure in an MPEG-2 file, one file per procedure. The software automatically discards outside-patient images. EM-Capture has been running at Mayo Clinic in Rochester, Minnesota since 2009 and captured over 71,000 anonymized endoscopic procedures thus far. The software itself is novel and eases the process of data collection significantly. We took part in developing the second application "EM-Automated-RT", which extends EM-Capture with other real-time analysis and feedback of quality of the colon exam. The second application was developed collaboratively by a team of seven reseachers and has been in used at Mayo Clinic Rochester since the 4th quarter of 2011.

This chapter describes our contribution in algorithms for real-time automatic detection of endoscopic procedures as well as software development for the two applications using SAPPHIRE and evaluation results.

6.1 Endoscopic Video Detection

In order to conduct quality control tests for every colonoscopy exam, we needed to digitally record and store a complete video of each individual procedure for later examination. The ideal system would be able to: (1) function without any user intervention (i.e., be transparent to all medical staff and physicians); (2) integrate well into the current medical infrastructure; (3) automatically turn itself on and off; (4) perform required analysis of the video stream in real-time; and (5) generate compressed video files. There were a number of challenges to overcome. First, complete system autonomy and integration with existing infrastructure varied from institution to institution. Second, analysis of colonoscopy video content was an underdeveloped area of research. Third, a combination of medical knowledge and expertise in computer science or engineering was necessary to develop the

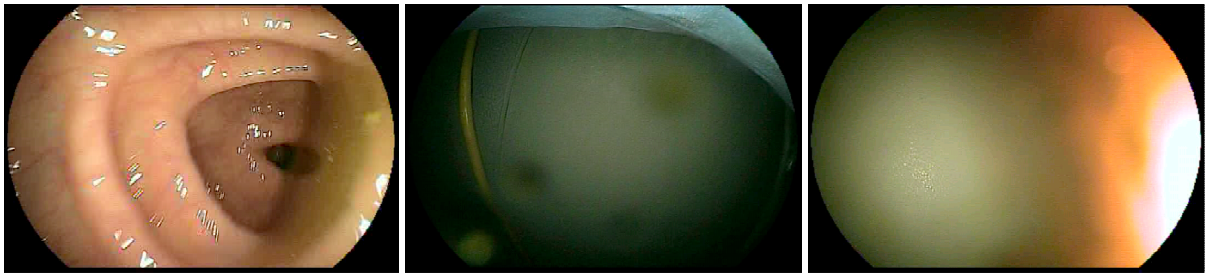

Figure 6.1: Examples of (a) inside-patient (left) and (b) outside-patient (middle) frames. An example of (c) an outside-patient frame (right) that resembles the color and brightness of an inside-the-patient frame due to an external light.

required algorithms. And lastly, developing and combining potentially dozens of algorithms into a single system while retaining a high level of accuracy was a complicated task.

As part of the first step toward objective quality analysis for endoscopy video, we introduced an automated colonoscopy video capturing and analysis system called “EM-Capture” that could automatically detect individual procedures and digitally compress and store these to the hard disk of a local workstation [74]. The videos were then uploaded from one or more workstations to a central server, where automated quality metrics could be performed. No people interaction was required at any step of the process. Since our system ran concurrently with actual colonoscopy exams as they occurred, our system needed to run in real-time to capture and analyze the video without dropping any important information. This was achieved through multithreading, using high or real-time prioritized threads, and assembly code for CPU intensive analysis. The accuracy of this step was extremely important, since a complete (i.e., not fragmented) video is needed to perform a comprehensive analysis reflecting the entire procedure. The solution we developed may also work – with minimal modifications – for other endoscopic procedures such as upper gastrointestinal endoscopy, cystoscopy, arthroscopy, and bronchoscopy.

We first tested the system in two endoscopy rooms at the Mayo Clinic in Rochester, Minnesota. Each room consisted of a completely separate set of hardware (including the workstations) for our software. Each workstation is a PC-compatible computer with a Pentium 4 CPU running at 2.8 GHz with 2 GB of RAM. Fujinon endoscopes were used for all procedures. The video signal from the endoscopes is converted to a digital signal for the computer by Turtle Beach Video Advantage USB video capture devices. Testing was done over the course of two weeks and totaled about 230 hours of raw video, with the system running from 6:00 am to 5:30 pm every day except weekends. It extracted 173 videos totaling 70 hours of recorded video, but contained 171 true procedures. The correct output of the system would be exactly one video file for each procedure. Improperly combining frames of multiple procedures into the same file is considered a false-join and improperly splitting a single procedure into multiple files is considered a false-cut. No videos had false-joins or false-cuts. Two video files did not have a procedure in them, but instead consisted of the camera laying on a table, pointed toward a bright light with a reddish hue or a white sheet of linen with a similar reddish hue. Although these types of videos are undesired, we prefer to capture more videos and not miss any procedures. Some videos contained, in addition to procedures, a period of time with the camera pointing toward a light shortly before or after the actual procedure. The extra images of such a light in a video are also undesirable, but obviously preferable to missing the beginning or end of a procedure by using too weak of thresholds.

We achieved segment-based sensitivity of 100% and specificity of 99% out of 173 videos. As it was difficult to determine the exact number of frames our software detected as being inside or outside a procedure, we calculated sensitivity and specificity based on the number of segments of video stream determined to be inside or outside a procedure. A true positive (TP) is a segment of video stream that the software correctly determined was part of a procedure. A true negative (TN) is a segment of video stream that was correctly determined to not be part of a procedure, and therefore was not captured. The TN number therefore consists of the summation of counts of all segments of video stream before, between, and after true positive and false positive videos. A false positive (FP) is a segment of video stream that was determined to be part of a procedure when in fact that segment of video

stream was not. A false negative (FN) is a segment of video stream that was part of a procedure that our system determined was not. Sensitivity is defined as TP / (TP + FN); that is, the percentage of procedures we were able to capture out of all procedures. Specificity is defined as TN / (FP + TN); that is, the percentage of segments of video stream that we did not record out of all segments of video stream that we should not have recorded. More details can be found in [74] and [75].

6.2 **Drawback of Old Method**

The previous method of splitting endoscopy videos apart involved a temporal analysis of the red, normalized-red, and motion metrics of the video stream. Although we achieved a high *segment-based* sensitivity and specificity (the original goal), we wanted to improve on the *frame-based* sensitivity and specificity.

A segment-based sensitivity separates the result data into segments of video containing inside-patient and outside-patient data. The transitions between inside and outside are recorded and compared against the ground truth transitions. Note that the exact time of a transition is not as important as attempting to find the transition from outside to inside either prior to or just at the real transition from outside to inside, and likewise from inside to outside. The main priority was to separate individual procedures into their own videos without missing any inside-patient data. The second priority was to eliminate as much outside-patient data as possible.

The previous method had very good results; but sometimes as much as forty minutes of outside-patient video data would exist in a video. As the intended use of the captured video would eventually be quality analysis by analyzing the video of each endoscopy, outside-patient data is undesirable. Additionally, outside-patient data can contain patient-identifiable images, which we must remove. With these new goals in mind, it makes more sense to use a frame-based sensitivity and specificity. That is, all video frames are classified as inside-patient or outside-patient frames and compared against the ground truth.

Using a frame-based sensitivity and specificity, the results of the previous method were not satisfactory. Although the sensitivity was still good (it had virtually 100% true

positives of inside-patient data and no false negatives), the specificity was as low as 80% (due to many false positives).

Also, while the thresholds of the old method worked well for the specific brand of endoscope we tested, they did not work well for different models or brands of endoscopes. Furthermore, video settings such as brightness, contrast, and tint greatly affected how well our thresholds worked. For example, enabling the tint button on one endoscope machine increased the normalized-red value for a frame by as much as 100. A white background could appear pinkish-red and possibly be detected as inside-patient data. To be robust, we need more flexible thresholds and potentially new video metrics.

6.3 New Metrics

To get better results, it is logical to find some metrics that differentiate the most between inside-patient and outside-patient video. For example, a metric that has a range of 0 to 100 and a value of 49 for outside-patient and 51 for inside-patient would not differentiate as much as a metric with the same range that had a value of 1 for outside-patient and 100 for inside-patient. Although it is possible that some frames could be classified as false-positive or false-negative using an individual metric, we want each individual metric to correctly classify as many frames as possible as strongly as possible. We choose different style metrics that strongly classify different kinds of frames, such that we would not choose several metrics that classified the same set of frames as false-positives, for example.

The three original metrics were very hardware and user dependent. We wanted to add metrics that would be able to adjust to the video regardless of hardware change or user settings. The metrics we added include: variance of middle 80% of the derivative of the mean-red signal over time, and the same for the mean-normalized-red signal; several histogram “energy” metrics of the mean-red, mean-normalized-red, and a special version of each containing filled histogram bins for every value between two successive data points; the mean-normalized-red rise and fall; and a long-term metric based on the energy histograms. Most of these metrics work well by “calibrating” to the data as it is seen – then when we transition from inside to outside or outside to inside, the metrics change drastically. The software including the new metrics was run in 8 endoscopy rooms at Mayo Clinic Rochester.

We first present our original EM-Capture application, followed by details on how we ported it to SAPPHIRE. EM-Capture consists of three separate major components that are run as a parallel pipeline: capture, analysis, and encoding. The capture component continually captures video from the video capture hardware and buffers it for the analysis phase. The analysis component analyzes each frame to determine the start and the end frames of a procedure. The encoding component writes only the inside-patient frames to video files on hard disk, with one complete procedure per file. As some components (each implemented as at least one application thread) are more response-time sensitive than others, we set specific priorities for each thread (e.g., video capture is of the highest priority).

6.4 Capture

The capture component captures audio and video data from the capture hardware, which receives the video from the endoscope hardware as a series of images, and stores it in a circular FIFO (First In/First Out) data buffer in memory (see Figure 6.2). Since features of a single frame are not sufficiently accurate to decide whether the frame is part of a procedure or not, a buffer (about ten seconds in our environment) is used to keep the most recent video data in memory. These frames stay in memory while being analyzed by the analysis component until a decision is made whether to use or discard the frames. For example, storing ten seconds of raw video data in 720 x 480 x 24 bits-per-pixel format at 29.97 frames per second in memory requires about 296 MB of memory. Since all threads in our system use the buffer, we allocate the buffer as a large chunk of shared memory. We ensure thread-safe access of this memory such that no component adversely affects another when it needs to read from or write to this memory. The buffer is used in a circular first-in first-out (FIFO) fashion, where new data is appended to the head end of the FIFO buffer as indicated by A in Figure 6.2. As newly captured frames become available, they will be analyzed in the same order they were captured. This corresponds to the frames between B and A in Figure 6.2. Once analyzed, a frame is either discarded if it is determined that it is not part of a procedure or saved in the buffer until a decision can later be made whether the frame is part of a procedure or not from analyses of later frames. In Figure 6.2, C points to the oldest buffered frame and the distance from C to B indicates the number of frames saved in this fashion. If

analysis on a future frame determines that the previously saved frames were not part of a procedure, these frames are discarded by repositioning C and D to point to B. As a result, all frames older than the frame pointed to by B are discarded. If the analysis of a frame determines that all the saved frames were in fact part of a procedure, all the saved frames need to be encoded and written to disk. The oldest saved frame that must be encoded to disk is the tail of the FIFO buffer, and is pointed to by D in Figure 6.2. Once frames are encoded and written to disk, they can be discarded, and the next frame can be encoded. This is done by advancing D one frame at a time toward C. However, D cannot advance past C since the frames still being buffered in C are not yet ready to be encoded. C cannot advance past B since the frames indicated by B have not yet been analyzed. B cannot occur before A since we cannot analyze frames that have not yet been captured. Finally, A cannot occur before D (in a circular fashion), since in that situation, the FIFO buffer would overflow, which would result in dropped frames.

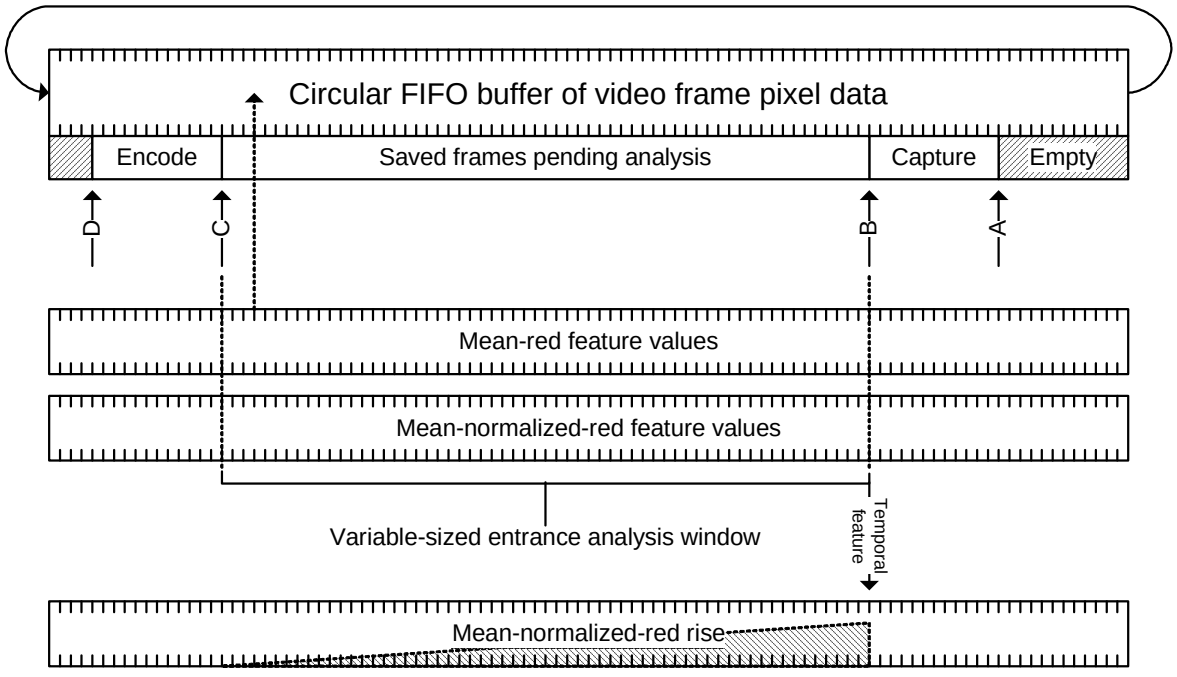

Figure 6.2: The circular FIFO video frame buffer contains several internal pointers based on how much each frame in the FIFO buffer has been processed. A indicates the head of the FIFO buffer where newly captured frames are written. B points to the oldest unanalyzed frame. C points to the oldest potential procedure image whose classification of inside or outside is still unknown. D points to the tail of the FIFO buffer where images are either written to disk or discarded. Each tick mark represents a single frame.

6.5 Analysis

The analysis component computes features from consecutive video frames in an analysis window. The output of the image analysis component is a value of a variable called inflag, which tells the video encoding component whether the image is part of a procedure (inflag = true) or not (inflag = false). Most procedures contain only images of the patient's colon mucosa for the entire duration of the procedure. However, occasionally, some procedures contain outside-patient images when the endoscope is briefly pulled out of the colon and re-inserted. A new procedure is started when inflag transitions from false to true, and a procedure is completed when inflag transitions from true to false. The actual start and end frames of a video file can be different from the exact frame of the transition, since, for example, we may elect to keep a few seconds before the actual procedure starts in order to see the insertion of the endoscope into the rectum (during testing of the software to verify that this is indeed the start of the procedure). The same applies for the end of the procedure where we can add a few seconds of video to provide evidence that the procedure indeed has finished.

The difficulty of the analysis is as follows. We are working with (1) an analog video source, which is prone to a large amount of noise, (2) a real-world environment where the contents of consecutive procedures can change significantly from procedure to procedure (e.g., upper-endoscopy followed by colonoscopy), and (3) the human factor, which presents many challenges such as changes in display settings (e.g., brightness, contrast, tint) by the physician, or the physician forgetting to switch on the video source of the endoscope or the endoscope light source until after the endoscope has been inserted into the colon (called a sudden start hereafter) when it is too late to calibrate or detect an entrance event. The latter two issues are the most difficult to deal with. We, therefore, designed novel features to achieve the following goals: (1) all procedures are recorded, (2) a very small percentage of false procedures are recorded, and (3) a tiny percentage of outside-patient images are included in the recorded procedure videos.

6.5.1 Characteristics of Inside-Patient and Outside-Patient Video

The video received from an endoscope may have an actual viewing area that is a rectangle, octagon, or part of a circle inside the video frame; pixels outside the viewing area are very dark, but not necessarily completely black (see Figure 6.1). These irrelevant pixels need to be discarded. Inside-patient images are primarily red or reddish-orange colored. Although there is usually a higher chance of the washed-out images (low amount of red) in outside-patient images, a high amount of red occasionally occurs when the endoscope points at some object (e.g., orange floor) very closely for minutes to an hour. Therefore, color features alone are insufficient. While inside the patient, the endoscope is constantly moving in and out as well as moving up, down, or sideways. Much more motion activity typically occurs during a procedure compared to between procedures. However, there are times of low motion inside the patient. For example, when pictures are being taken for reporting purpose, or biopsy/excision material is being inserted or withdrawn through the endoscope, the endoscope tip may not move for seconds to several minutes. Once a procedure has been completed (i.e., the endoscope is outside the patient), there usually is very little motion since most of the time the endoscope lays stationary on a tray or there is no video signal as the endoscope is unplugged from the video processor. However, high motion occurs occasionally (e.g., the endoscope has been removed but not set down on a tray yet). Additionally, there are often times where video noise is significant enough in that even though the camera is still and the objects are still, the noise causes significant motion.

Given the real-time constraints and the above characteristics of the input video signal, we use two sets of features: (1) basic features to discard obvious irrelevant pixels or outside-patient frames and (2) our new temporal features derived from change in certain information among consecutive frames over time to deal with more complex cases. To derive frame features, we first discard pixels that are unlikely part of the endoscope viewing area as follows. We use a conditional filter to accept only pixels whose red, green, and blue components are all at least a mucosa threshold value. Pixels that do not meet the criterion are treated as invalid pixels and completely ignored by the analysis algorithms. If less than the minimum area threshold of the video area contains valid pixels, some features (e.g., mean-

red and mean-normalized-red to be discussed shortly) are also set to zero, since we consider this frame to contain an insignificant amount of information to be part of a real procedure.

As frames are analyzed using the actual frame data in the video FIFO buffer (Figure 6.2), values of basic features and temporal features are generated. Each feature value is usually represented by one or more number, which is significantly smaller than the video frame data. These temporal features are either derived from the video frame data in the time window or the basic features in a different time window. Some temporal features (e.g., variance of differences of mean-normalized-red values) require a constant-sized buffer while other features (e.g., mean-normalized-red rise area) operate on a variable-sized window of basic feature values. This variable-sized window may correspond to the video frame data stored in the circular FIFO buffer between C and B in Figure 6.2. Some other features may operate on their own window of recent frames.

6.5.2 Basic Features

Let I be the image for a given frame, with each pixel having a red ($I(x,y).r$), green ($I(x,y).g$), blue ($I(x,y).b$) component values between 0 and 255 in RGB color space, and n is the number of pixels whose r, g, b values are all at least mucosa threshold.

Mean-Red, Mean-Normalized-Red, and Accumulated Mean-Normalized-Red

The mean-red for each frame is the average red intensity value in the RGB color space of all valid pixels over a single frame of video as shown in Equation 1. The mean-normalized-red (mean red saturation) calculated using Equation 2 is the amount of red saturation within each pixel (compared to green and blue), averaged over all valid pixels in a single frame. Red saturation is useful to detect dark red, where the mean red would give too small a value, and to exclude red containing colors such as white, which has equal quantities of green, blue, and red. The accumulated mean-normalized-red (AMR) is defined in Equation 3. AMR is initialized to zero and is occasionally reset based on the inside-outside detection logic discussed in Section 6.6.

These features are simple and extremely fast to calculate, which makes them suitable for real-time applications. However, they are ineffective in many situations. For example, a

camera pointed at a bright white cloth or some close object in the room with a bright reddish tint will have a very large mean-red value, which does not distinguish it very well from being inside the colon. If a physician changes the video settings (e.g., tint), the amount of red saturation for an ambient outside-patient video frame could be unnaturally high; for instance, when the endoscope is pointed toward a white wall in the procedure room, a solid white wall may instead look like a solid pink wall. The values of mean-red for outside-patient images from our experience vary from 10 to 252 (although skewed toward the lower end) and inside-patient images from 30 to 255 (usually skewed toward the higher end).

$$mean\text{-}red = \frac{1}{n}\sum_{x,y} I(x,y).r \tag{6.1}$$

$$mean\text{-}normalized\text{-}red = \frac{1}{n}\sum_{x,y}\left\lfloor \frac{255 \cdot I(x,y).r}{I(x,y).r + I(x,y).g + I(x,y).b} \right\rfloor \tag{6.2}$$

$$AMR = \sum_{time\ \text{window}} mean - normalized\ - red \tag{6.3}$$

Histogram Difference

To measure the amount of motion, there are several complex methods such as optical-flow or block-based motion estimation [76]. However, these methods are too time-consuming for real-time applications. In our previous work, we elect to use a variation of the chi-squared histogram difference [69] to capture the motion between two frames. This feature is no longer used in this work since we have new temporal features that provide better performance; however, it is mentioned for completeness.

6.5.3 New Temporal Features

The basic features are insufficient for achieving the three aforementioned goals for the analysis. Thus, we introduce new features that all rely extensively on temporal information rather than just on one or two frames worth of information. We calculate our new features using the video already in the buffer.

Mean-Red and Mean-Normalized-Red Variance of Differences Without Outliers

Figure 6.3 shows that the video before and after a procedure typically has little variation in its mean-red and mean-normalized-red values, but the video during a procedure

has a high amount of variation. In order to capture variation of mean-red values over a longer duration, we can calculate the variance of mean-red values and mean-normalized-red values over a number of frames. However, this simple variance cannot readily distinguish the start and end of a procedure. Instead, we use the variance of the differences of mean-red values of a pair of consecutive frames for all *w-1* pairs of *w* consecutive frames in the video buffer after removing the outliers (the mean-red differences outside the 10 and 90 percentiles of the differences). The outliers are due to the scene changes (e.g., endoscope repositioned from dimly lit far away wall to nearby brightly lit cloth). Let *K* be the set of the mean-red differences without the outliers and $|K|$ denotes the size of this set. Let $DiffMeanR_i$ be the i-th mean-red difference in the set. We calculate the mean of the mean-red differences and the mean-red-difference variance using Equation (6.4) and Equation (6.5), respectively.

$$\overline{DiffMeanR} = \frac{\sum_{i \in K} DiffMeanR_i}{|K|} \tag{6.4}$$

$$DiffVariance = \sum_{i \in K} \left(DiffMeanR_i - \overline{DiffMeanR} \right)^2 \tag{6.5}$$

Similar equations are used to calculate the variance of the mean-normalized-red differences by substituting the mean-red difference with the mean-normalized-red difference. In a real endoscopy video, when we remove the outliers and calculate the variance this way, the variance at the start and end of a procedure is non-zero and is significantly higher than the variance during a procedure (usually non-zero). The variance during the procedure is typically higher than the variance for the outside–the-patient images, which is zero, or very close to zero. Figure 6.3(b) demonstrates the effectiveness of this feature.

While this feature works well to correctly remove a large number of outside-patient frames that would have otherwise been recorded, there are four problems. The first problem, mentioned before, is that when the endoscope is inside the patient but idle, the variance will be zero. The second problem is that occasionally continuous motion changes of the endoscope while it is outside-patient also have high variance. The third problem is that a “sudden start” procedure (a procedure that begins inside the colon – e.g., the video feed was

not plugged in until after the endoscope was inserted) may be difficult to detect using this feature alone, since the scene change from solid black (or very dark) to endoscopy frame signal does not produce as high variance as we normally see during a regular colon entrance. The fourth problem involves noise in the video capture. Not infrequently, the mean-red and mean-normalized-red values can rapidly oscillate in intensity from one frame to the next (similar to a square wave from 0 to 255). Sometimes, the oscillation is less predictable and occurs every few frames. If we rely on differences between frames, the differences themselves are similar to a square wave, resulting in very large variance. We, therefore, propose another feature to address this oscillation problem.

Mean-Red and Mean-Normalized-Red Energy Histogram Area

In the simplest case, the oscillation problem mentioned above will have two values between which the mean-red or mean-normalized-red oscillates. We address this problem by creating a histogram of all the data points (mean-red or mean-normalized-red) within a time window, with one bin per integer intensity between 0 and 255. We call this histogram an “energy histogram”. With a simple square-wave type oscillation between 0 and 255, our energy histogram has two non-zero bins (0 and 255). In a less trivial example, there could be several non-zero bins. In an endoscopy procedure, as the endoscope moves, the mean-red and mean-normalized-red values generally rise and fall quickly and often throughout most of the video. Thus, a procedure’s histogram should have many bins filled in seemingly random amounts. If our energy histogram has only a few bins filled, or a few bins containing the vast majority of data points, it is very likely that the video segment is not part of a procedure.

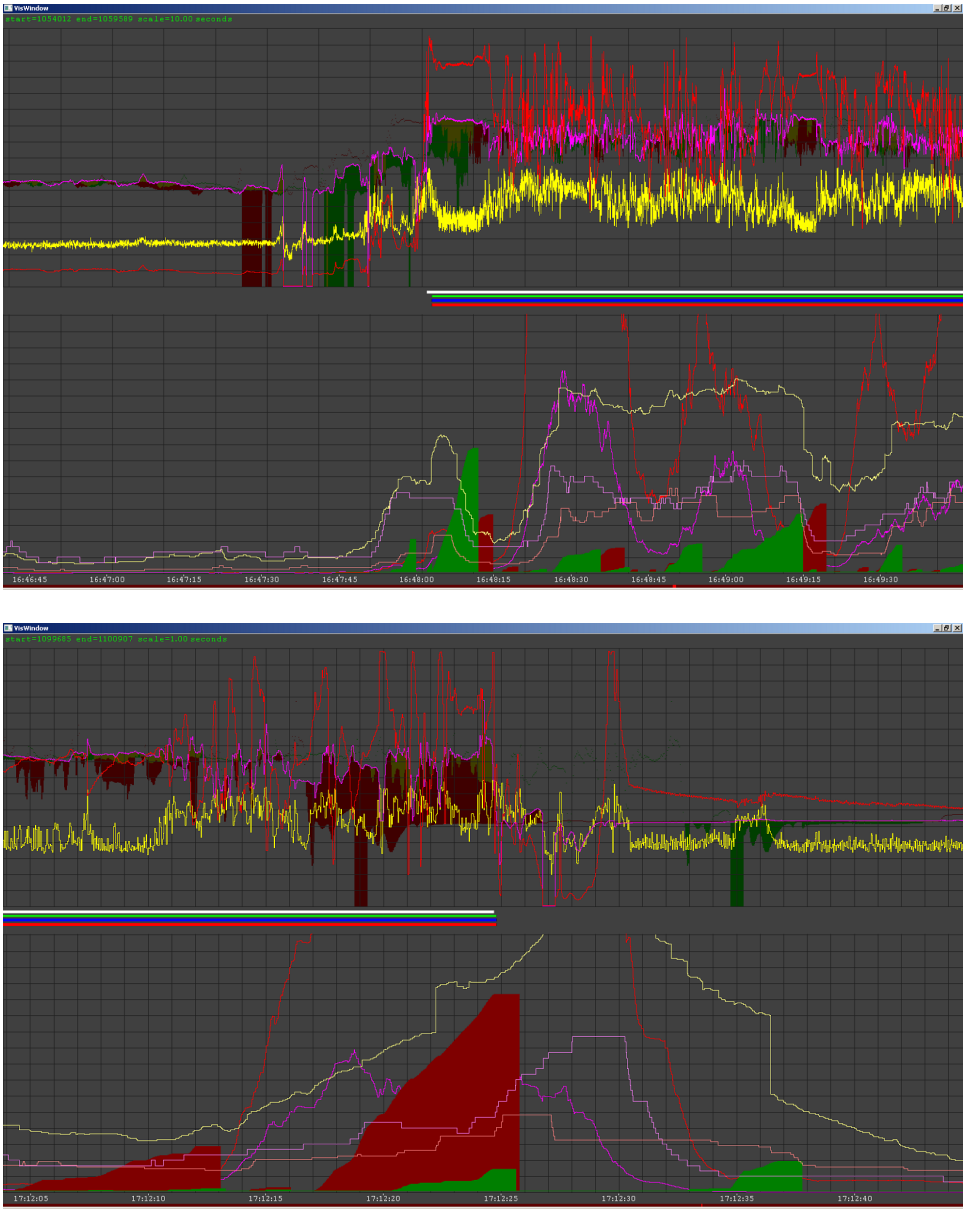

Figure 6.3: Examples of features graphed over time of (a) a sample procedure entrance, shown top, and (b) exit, shown bottom. In the top half of each image, the red line corresponds to the mean-red feature, the magenta line to mean-normalized-red, the yellow line to the histogram difference; and the green and red areas correspond to the mean-normalized-red rise and fall differences, respectively. On the bottom half of the figure, the red line corresponds to the mean-red variance of differences, the magenta line to mean-normalized-red variance of differences, the tan line to mean-red energy histogram, the pink line to the mean-normalized-red energy histogram, and the light yellow line to the hybrid mean-red/mean-normalized-red energy histogram. In the middle, the red, green, and blue bars correspond to frames being detected as frames inside a procedure by our algorithm. The white bar is the ground truth for frames being part of a procedure, set by a real person observing the actual video and marking the precise entrance and exit frames. A sharp rise in mean-normalized-red, one of the features shown on the top half of each graph, is indicative of the precise entrance frame.

Figure 6.4: Energy histograms generated from two different sets of data. For the left histogram, a very high peak of one value causes the other bins to be scaled to a smaller value. The high peak is likely due to a flat signal, indicating very little change, and usually only occurs outside a procedure. The "area" computed is the space filled (the histogram bins) within the overall histogram graph (the yellow box). When a high peak causes the rest of the values to be scaled very low, the overall area will also be low. For the right histogram, we have more regular variation in the mean-red values. As a result, more of the histogram bins are allowed to retain a higher value when scaled. The overall area occupied by the bins in the histogram will be significantly larger in this case. This usually occurs during a procedure.

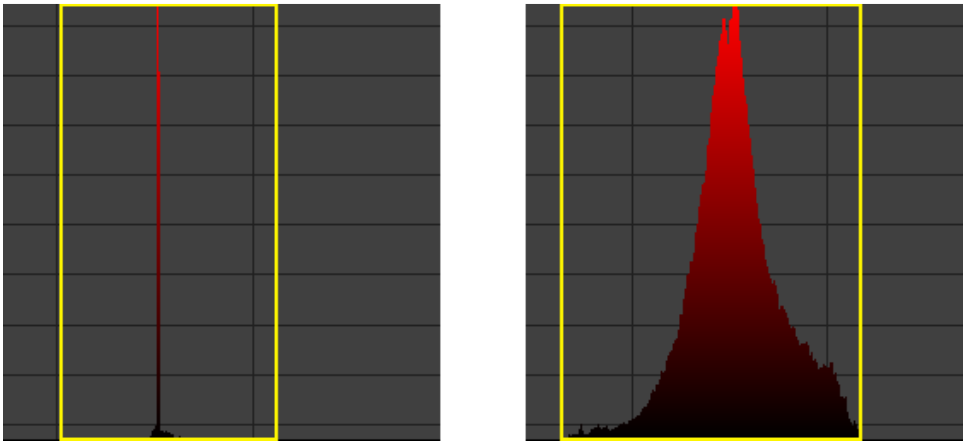

Figure 6.5: Double-normalized energy histograms generated from two different sets of data. For the right histogram, a typical signal of a procedure has its "area" (the space taken by the actual bin data) computed within a rectangle whose left and right bounds are the first nonzero bin and the last nonzero bin, respectively. This is shown as the yellow rectangle. We normalized the area computed inside the rectangle, as if the area inside the yellow rectangle were 1.0. This is different than the previous feature in that the width was always 256 bins, whereas in this feature, the width adjusts to the data. For the left histogram, a somewhat flat signal outside the patient has resulted in a large peak, similar to the histogram in Figure 6.4. Although the rectangle that can fully encompass the nonzero bins can be much smaller than the yellow box indicates, we restrict the minimum width of the histogram to 128 bins; otherwise, this peak could cover a considerable area in the resized boundaries, generating a high value.

To derive a useful feature from the energy histogram, we normalize the histogram and calculate the "area" of the normalized energy histogram as follows. The largest bin count in the histogram is scaled to 1.0, and all other bin counts are scaled similarly with respect to the largest bin. The width of the histogram is scaled from 256 to 1.0 as well, so that the "area" of the histogram becomes a unit square of 1.0. We sum up all of the scaled bin count values and divide by 256, resulting in the "area" of the normalized energy histogram. A smaller area means that few bins contained most of the data points, as they appear as large spikes in the histogram, and other bins become insignificant. Summing the maximum bin count value for just a few bins results in a small area. If, on the other hand, there is a good spread of data points, many bins have a significant non-zero value, and the area of the histogram is much larger. This feature successfully addresses the oscillation problem.

Double-Normalized Mean-Red Energy Histogram Area

This is derived from the previous feature for detecting a sudden start. This feature involves normalizing the number of bins to the width of observed values. Instead of calculating the area of the histogram as a 1.0 by 1.0 square, from bins 0 to 255, we calculate the area as a 1.0 by 1.0 rectangle from bins X to Y, where X and Y are the minimum and maximum indices of nonzero bins. The width of the histogram is considered from the smallest observed value to the largest observed value within the histogram. Thus, if for some window of time, we never see a mean-red value below a certain threshold, for example 60, we normalize the histogram using that value as a lower bound for calculating the area. The maximum area value in this case would be normalized to 1.0 and allows for a range of widths of histograms. To prevent small fluctuations (such as the simple oscillation problem) from having high areas, we restrict the minimum width of the histogram to 128 bins (half of the histogram). We have empirically observed that this variation is very effective for finding sudden starts within a one-minute window. Although not ideal, since we already miss some data in a sudden start and additional data in the one-minute window, we will at the very least begin capturing some of the otherwise possibly completely missed video.

Hybrid Mean-Red/Mean-Normalized-Red Energy Histogram

This feature counts the number of bins with the bin count above a specific threshold (e.g., the bin count is at least 5% of the largest bin). While similar in nature to the previous feature, this feature is less sensitive to the height of each bin (which contribute to the area), and is more dependent on how many bins have a significant number of samples.

All the features presented thus far are not able to find precise entrance and exit frames. They all work in a window of time and can provide only the general vicinity for an entrance or exit. The following features help to detect the precise entrance or exit.

Mean-Normalized-Red Rise/Fall Area (Entrance/Exit)

Figure 6.3(a) shows that the mean-normalized-red rises very suddenly at the precise entrance for most videos, which can be used to detect the precise entrance. Features based on differences of successive input data points over time (like a derivative) are not very useful since noise in the signal causes many positive and negative values to occur. We shift the mean-normalized-red signal by *w* frames forward in time and subtract the shifted signal from the original signal. The area between the plots of the two signals is analogous to the sum of the differences over a window of time. In other words, we compute the difference at each data point as the value of the original mean-normalized-red at that point less the value of the shifted mean-normalized-red at the same point. We compute the sum of the differences in a time window as follows. Initially, the sum is zero. We add to this sum if the difference is above a threshold value. If the difference is negative and smaller than a threshold value (representing a fall instead of a rise), we reset the sum. At the frame when the value of the sum goes over a threshold value, we determine that the frame is an entrance frame, as seen in Figure 6.3(a). Video encoding starts recording at this frame.

To determine an exit instead of an entrance, we derive another feature by shifting the signal backward in time by *w* frames and subtracting the shifted signal from the original signal. If the shifted signal is smaller than the original signal, a fall in the mean-normalized-R signal occurs. We compute the cumulative sum representing the area as follows. We reset the sum if it is above a threshold value and add to the sum if the difference is smaller than a

threshold value. When the value of the sum becomes lower than a threshold value, we determine that the frame is an exit frame, as seen in Figure 6.3(b).

Because there may be several rises and falls before, during, and after a procedure, this feature must rely on other features first for locating the general vicinity of an entrance and exit. When given that approximate location, we utilize these rise and fall features to pinpoint the entrance and exit frames to within fractions of a second. This feature is very effective in removing every outside-patient image and including every inside-patient image.

Because all features are computed continuously, the rise and fall features are available whenever a candidate entrance or exit is detected. There may be many rise/fall areas, even before or during a procedure. The specific rise or fall area used may be chosen due to the other features being strong enough to indicate an entrance. The start of a rise or the end of a fall indicates the frame of entrance or exit, whereas the other features only indicate that a procedure has started/finished within some given time frame.

Unfortunately, there are some drawbacks to this method. In particular, it requires several outside-patient frames and inside-patient frames to derive the features. If these frames do not exist (e.g., the sudden start scenario), these features are not reliable. To improve the overall accuracy of this feature, we exclude rises when the video signal starts at zero (black). This is to prevent the signal generated by an endoscope that has just been plugged in from being considered a procedure, since it usually is just displaying outside-patient data (which is clearly a rise from black). Unfortunately, a side effect is that if we plug in the endoscope after insertion, we cannot determine immediately whether the camera is inside or outside the patient. Instead, we must rely on the other features. Another drawback, which occurs rarely, is that the rise is very subtle. Since the primary requirement of the system is to record every procedure and miss no procedures, we are forced to use a low threshold to avoid missing these rare cases, which in turn increases the false positive frame capture rate.

6.6 Algorithm for Identifying the Start of a Procedure

Figure 6.6 shows our algorithm that utilizes the above features to determine the entrance frame. The algorithm sets the value of inflag as either true (the camera is inside the

patient) or false (the camera is outside the patient) and determines the start frame of a procedure. We set the initial state of inflag to false. We keep a variable-sized window to retain frames in memory to compute temporal features from these frames. Step 1 in Figure 6.6 resets the variable-sized analysis window if the specified conditions are met. Step 2 detects different entrance scenarios. For a sudden start, we generally cannot rely on many features since they need to calibrate on the outside-patient video to detect the outside-to-inside transition. For a normal procedure entrance scenario, several features are used together for accuracy. For mean-red variance, mean-normalized-red variance, and mean-normalized-red rise area, each of these features has two sets of thresholds. This is because some procedure videos vary considerably in brightness, contrast, tint, and quality. The threshold for the accumulated mean-normalized-red is either set to require a small amount of red saturation for each frame within the past analysis window of video analyzed, or a much larger amount of red saturation within a shorter time period. The list of thresholds and values used in our experiments is shown in Appendix H. As part of our future work, we will use a machine-learning classification algorithm to find optimal threshold values. Step 3 sets values to indicate the transition to the “inside-patient” state. The precise start frame of the procedure is the earliest frame used in the mean-normalized-red rise area (i.e., the frame following the last time the sum was reset). The analysis component then sets *inflag* to true and signals the video encoding component to begin processing at that start frame. The processing continues until the system has decided that the endoscope has been fully removed from the patient.

Step 1: Check conditions for resetting the analysis window size to zero.

1. Area of the double-normalized mean-red histogram below its threshold
2. One of the following is true. a) Mean-red below its threshold; b) Mean-normalized-red below its threshold; c) Variable-sized analysis window above a window size threshold; or d) Mean-normalized-red difference (calculated from one time step of the mean-normalized-red rise area feature) below its difference threshold.

If both conditions 1 and 2 are satisfied, reset the variable-sized analysis window size to zero and reset the accumulated mean-normalized-red to zero.

Step 2: If *inflag* is false and the below condition for transition from outside to inside is satisfied, go to Step 3; otherwise, skip Step 3.

The transition happens if either the area of the double-normalized mean-red histogram above its threshold (a sudden start case) or all of the following are true. a) Mean-red above its threshold; b) Accumulated-mean-normalized-red above its threshold; and c) At least one of the following conditions is true:

1. Either all of the following are true:
   - Variance of mean-red differences above its *high threshold*
   - Variance of mean-normalized-red differences above its *high threshold*
   - Mean-normalized-red rise area above its *low threshold*
   - Mean-red energy histogram above its threshold
   - Mean-normalized-red energy histogram above its threshold
2. Or all of the following are true:
   - Variance of mean-red differences above its *low threshold*
   - Variance of mean-normalized-red differences above its *low threshold*
   - Mean-normalized-red rise area above its *high threshold*

Step 3: Do all of the following.

1. Set *inflag* to true (transition to inside the patient)
2. Mark the first frame of the variable-sized analysis window of frames as the actual entrance frame
3. Set the accumulated-mean-normalized-red to zero

Figure 6.6: Algorithm to detect the entrance frame; the threshold values are based on training.

6.7 Algorithm for Identifying the End Frame of a Procedure

The inter-procedure duration varies depending on the start time of the next procedure. If the next procedure is on a different patient, the inter-procedure duration is long since the current patient needs to be removed from the room followed by the new patient entering the room. The inter-procedure time is typically shorter if the next procedure is on the same patient (for instance, colonoscopy followed by upper endoscopy). We keep a running total of the number of potentially outside-patient frames since the first candidate end frame is identified. If we find that at least a certain percent of the past five minutes are identified as outside-patient frames, we have a high confidence that we have found the end of the

procedure. The duration of five minutes is based on domain knowledge that the minimum time between procedures is at least five minutes.

Algorithms for identifying the end frame cannot rely on detecting low motion alone since the content of a frame may be same for a few seconds to minutes even though the endoscope is inside-patient, for example, when the endoscopist carefully examines a specific region of the colon, takes a picture of a frame, or waits for specific equipment not immediately available. The amount of motion can be very low during such events.

Step 1: If all of the following are true, flag current frame as a potential outside-patient frame; otherwise, flag it as an inside-patient frame:

1. Area of the double-normalized mean-red histogram below its threshold
2. Any of the following is true:
   - Variance of mean-red differences below its threshold
   - Variance of mean-normalized-red differences below its threshold
3. Any of the following is true:
   - Mean-red below its threshold
   - Mean-normalized-red below its threshold
   - Hybrid mean-red/mean-normalized-red-energy-histogram-area below its threshold

Step 2: If either of the following is true:

1. Area of the double-normalized mean-red histogram above its threshold
2. All of the following are true:
   - Mean-red above its threshold
   - Accumulated mean-normalized-red above its threshold
   - Variance of mean-red differences above its threshold
   - Variance of mean-normalized-red differences above its threshold
   - Hybrid mean-red/mean-normalized-red-energy-histogram-area above its threshold

Then, do all of the following:

1. Reset the running total of potentially outside-patient frames to zero
2. Reset the position that an exit might occur to the next frame
3. Reset the accumulated mean-normalized-red to zero

Step 3: If the running total of flagged potentially outside-patient frames is at least four and a half minutes worth, transition to an outside-patient state and continue to Step 4. Otherwise, skip Step 4.

Step 4: Find the exit frame: Scan backward from the current frame until we find the exit frame in which all of the following are true at that frame.

1. Mean-normalized-red fall area above some threshold
2. Mean-red variance of differences above some exit threshold
3. Hybrid mean-red / mean-normalized-red energy histogram area above its threshold

Figure 6.7: Algorithm to detect the exit frame; the threshold values are based on training.

The algorithm is shown in Figure 6.7. Step 1 flags individual frames as inside-patient or outside-patient. We maintain the running total of the number of potentially outside-patient frames over the five minute window mentioned. This is used later to determine whether we should transition to outside-patient. Step 2 attempts to avoid a prematurely detected exit by continually checking every frame for detection of the outside-to-inside transition. If at any frame, we would have made that transition to the inside state (ignoring the fact that we are already in the inside state), we reset the running total, reset any potentially outside frames as instead being inside-patient, and reset the first potential end frame to the current frame – we know with high certainty that those frames are still those of inside-patient. The combination of both of these steps is more robust than either step alone. Step 3 makes the final decision about whether or not to transition to the outside state based on the number of frames that were marked as outside-patient. Step 4, finally, identifies the precise exit frame. In the best case, we find the exit frame exactly as the ground truth. In the worst case, the detected exit frame is after the actual exit frame. In other words, we include extra frames after the procedure has ended. This is to avoid missing the end part of the procedure in which an important quality indicator such as retroflexion typically occurs. Figure 6.7 shows the details of the algorithm. The algorithm is threshold-based by design to ensure that the analysis can be done in real-time to prevent frames in the FIFO buffer from being overwritten.

6.8 Video Encoding

Once the analysis component decides the start frame of a procedure, the video encoding component creates a new MPEG-2 video file and writes the audio and video data available to the file. After each individual video frame is written to the MPEG-2 file and no further analysis of that particular frame is required, its space in the FIFO buffer is marked as free, and the video capturing component may safely overwrite it. Video encoding continues until the analysis component has determined that the next frame in the video buffer is no longer part of a procedure. When this happens, the video file is closed, and the process repeats when the analysis component determines that a new procedure has started. As explained so far, the first frame that is encoded in a video file is the first frame of the window

mentioned in the “entrance” step, and the last frame that is encoded is a frame four and a half minutes after the determined “exit” frame.

For the end of a video file, we have only specified that the procedure ends four and a half minutes prior to the actual time that the analysis component determines as the end of a procedure (to make sure we avoid reentrance conditions where we might accidentally split a single procedure into multiple procedures). Since we only buffer ten seconds worth of video data at any given time, these four and a half minutes have already been written to disk by the time we determine the procedure has ended. We can delete the outside-patient frames written to disk by seeking four and a half minutes back from the end of the MPEG-2 file and truncating the file at that point. The four and a half minutes are based on the domain knowledge that the time between procedures is at least five minutes, and we allow a misdetection rate of 10%.

Another component of interest is a Windows system service that we developed to automatically start and stop EM-Capture at specific time points. We configured our capture system to run from 6:00 am to 8:00 pm every day except weekends. The service also started EM-Capture if the computer is turned on between these hours. We use a system service instead of a standard Windows task scheduler event to execute the software automatically when Windows has loaded but no user has logged in. Once installed, no user intervention is required for our software.

6.9 Experimental Results

We derive the values of the thresholds from experiments with a training image set recorded prior to our experimental testing image set. These thresholds are shown in Appendix H. We then tested the system in eight endoscopy rooms at the Mayo Clinic Rochester. Each room consisted of a completely separate set of hardware (including the workstations) for our software. Each workstation was a PC-compatible computer with a Pentium 4 CPU running at 2.8 GHz with 2 GB of RAM. Fujinon endoscopes were used for all procedures. The video signal from the endoscopes was converted to a digital signal for the computer by LeadTek WinFast TV2000 XP Expert video capture devices. Testing was done over the course of one month and totaled about 2,464 hours of raw video, with the system

running from 6:00 am to 8:00 pm every day except weekends. The correct output of the system would be to create exactly one video file for each procedure with all outside-patient frames removed from each video file. We implemented visualization software to assist frame classification. We classified all 2,464 hours of video (over 265 million frames) as inside-patient and outside-patient for each frame.

Since there are no other existing works that perform the same task, we only evaluated our proposed work against our previous work. Using our previous method, no videos had false-joins (two different procedures are put in the same file) or false-cuts (one procedure divided into more than one files), but there were several false positive videos. These false positive videos consisted of the camera lying on a table, pointed toward a bright light with a reddish hue (as seen in Figure 6.1) or a white sheet of linen with a similar reddish hue. Some videos contained, in addition to procedures, a period of time with the camera pointing toward a light shortly before or after the actual procedure. The extra images of such a light in a video are also undesirable, but obviously preferable to missing the beginning or end of a procedure by using too stringent thresholds. As much as an hour of extra outside-patient data has been captured in some videos using our previous method. The previous method on this data set had frame-based sensitivity of 100.00% (all inside-patient images were captured), but only a specificity of 89.22% (outside-patient images were misidentified as inside-patient images). With the proposed work, we achieved a sensitivity and specificity of 99.90% and 99.97%, a significant improvement. We missed a very small portion of the beginning or the end of some videos while some videos have a few extra frames. The missed frames are the blurry, uninformative frames that are common at the beginning and end of endoscopy videos, containing a reddish-orange gradient similar to Figure 6.1(c). While the entrances and exits were determined to include these kinds of frames, they are not actually useful for analysis. We do not miss informative images, such as the retroflexion images at the end of a procedure. While still satisfying the primary objective of not missing any informative frames of a procedure, we significantly improved on the secondary objective of removing outside-patient frames. Table 6.1 summarizes this result.

Table 6.1: Effectiveness of image-analysis methods for inside/outside patient image classification for over 265 million frames

	Frame-based sensitivity	*Frame-based specificity*
Previous method	*100.00%*	*89.22%*
New method	*99.90%*	*99.97%*

The system successfully ran in real-time, with the analysis pipeline only using about 25% of the CPU time available – about 8ms of computation time per frame (up from 15% CPU time of our previous method), the capturing component using about 1%, and the MPEG-2 encoding component varying its quality to use whatever remaining CPU time. Encoding MPEG-2 video at the maximum possible quality in real-time can easily take more than the maximum computing power of the workstation. Therefore, the video encoder changes the quality settings as it encodes each frame, depending on the amount of available CPU time. This allows our system to attain the highest possible MPEG-2 compression quality for the amount of CPU time remaining after all other computation. CPU time is calculated by using a software performance and optimization package to measure the amount of time spent inside specific functions and comparing this with CPU time spent inside the whole program. The entire process constantly utilizes about 99% of the computer's total available CPU time during video encoding, making full use of available processing time. No video frames were dropped during testing, with 100% of frames going through the analysis pipeline, and potentially available for video encoding.

6.10 Porting EM-Capture to SAPPHIRE

EM-Capture originally consisted of 4 threads, using image analysis algorithms to distinguish inside-patient and outside-patient frames, and ran in real-time (processed at least 29.97 frames of video per second). It was not easy to add additional processing components or update any part of the application, as the processing and communication pipelines were explicitly coded. To insert a new image analysis stage in the processing pipeline, we had to add another communication layer between the old analysis stages and the new analysis stage.

Additionally, if we wanted to give the new stage its own execution thread, we needed to add more thread-safe synchronization code to support this.

To port EM-Capture to our middleware, we broke apart the program into separate tasks for each major analysis component. Common data types were created for each module to read and write (e.g. MEANR, MEANNORMR), and to functionally understand what the data means. The original code for each task was inserted into a skeleton module (provided by the middleware) into a new function, and each module's Data callback function was modified to call this function. Very little code was actually changed; in fact, a significant amount of code was removed, since we no longer needed to explicitly perform the communication and synchronization as part of our code. Still, a small overhead of additional code was needed to wrap the old code with the new skeleton and middleware API.

The videocapture and/or mpegreader modules provide the stream of source data (RAW_VIDEO) and the autoresize module crops this data and outputs the VIDEO. Functionally, this is very similar to the original EM-Capture, except that with SAPPHIRE, this is done implicitly and safely multithreaded and all communication is handled by the middleware. About ten image analysis modules were created to use this data and output their own metrics through packets. Originally, many of these image analysis components were performed by a single module and in serial. The new EM-Capture using our middleware consisted of 28 threads, as compared to only 4 threads in our original program, yielding a much more balanced distribution of load between processors, as well as being able to utilize more than only 4 processor cores, if more were available.

Instead of bottlenecking on one thread, which caused offline processing using MPEG2 video files as the source stream (i.e., playing a video file at maximum speed and processing input data as fast as possible) to work at about 40 frames per second on average, we are now able to process at about 90 frames per second on average on the same computing hardware. We validated the correctness of the new version of EM-Capture by comparing the output metrics computed over hundreds of hours of video with the original version.

6.11 Case Study 1: EM-Capture (Procedure Detection)

The original version of our EM-Capture application (before using SAPPHIRE) ran in 8 procedure rooms of Mayo Clinic in Rochester, MN since 2007. This was expanded in 2009 with the beta testing of a SAPPHIRE-enabled EM-Capture to another 5 rooms. As the stability of the new version was satsifactory, we converted all of the 13 procedure rooms to use SAPPHIRE. In total, as of April 2012, EM-Capture has analyzed roughly 50 billion frames of video, and from that, successfully detected and recorded over 71,000 procedures in a real hospital setting. EM-Capture has been a very successful proof of concept program, successfully being ported to SAPPHIRE with minimal effort, while gaining all the benefits that SAPPHIRE offers.

6.12 Case Study 2: EM-Automated-RT for Real-Time Feedback

This case study application is motivated by the need to improve quality of colonoscopic procedures. The American College of Gastroenterology (ACG) and the American Society of Gastroenterology (ASGE) in 2006 published consensus objective guidelines defining a good quality colonoscopy. The guidelines for a screening colonoscopy after age 50 include: (1) a withdrawal time for patients without symptoms and with intact colon anatomy of at least 6 minutes; (2) documentation of visualization of anatomical landmarks such as appendiceal orifice and/or ileocecal valve in the cecum; and (3) an average polyp detection rate (the percentage of patients with polyps detected during colonoscopy) in male and female patients greater than 25 and 15 percent respectively [63]. However, there were previously no computer-aided methods to measure quality of colonoscopic procedures as recommended by the ACG.

Using EM-Capture as a base to detect endoscopy procedures, a larger group composed of many researchers worked collaboratively to develop EM-Automated-RT – an application for real-time quality analysis and feedback of colonoscopy video, using SAPPHIRE. EM-Automated-RT involved the creation of several modules listed in Table 6.2. A full list of modules, their description, and their inputs/outputs is available in Appendixes B and C.

Table 6.2: Modules added to create EM-Automated-RT

qcmetricrt.dll	*Quality metric reporting*
blurry.dll	*Blurry frame detection*
egd.dll	*EGD detection*
retroModule.dll	*Retroflexion detection*
rteoi.dll	*Real-time end-of-insertion detection*
rteoi_user.dll	*Manual end-of-insertion*
rts.dll	*Real-time stool detection*
spiralcounting.dll	*Spiral counting*

The real-time feedback program has several goals in mind: (1) detect blurry frames to distinguish informative and non-informative frames for other modules, while also reporting a colonoscopy quality metric of percentage of blurry frames; (2) detect various levels of stool during the procedure for quality reporting purposes; (3) detect end of insertion of endoscope in the colon to determine insertion time, after which the withdrawal phase begins, which is usually the most important phase of a procedure; (4) endoscope spiral motion counting by lumen detection and quadrant coverage, which is displayed as real-time feedback over top the endoscopy video during real medical procedures. When the lumen (dark area in an endoscopy video that represents the empty tunnel of the colon) is seen in a particular quadrant of the video, that quadrant is marked. Marked quadrants are shown as video feedback with a green triangle in that quadrant's corner of the screen. When all four quadrants have been marked, the spiral score increments by one and the marks are cleared.

It is proposed that the spiral motion of the endoscope is indicative of looking at an increased amount of colon wall (tissue), which is where colon polyps and cancers develop. If the endoscope is facing the lumen directly, it will appear in the center of the video. Thus, minimal view of the wall is achieved. When the endoscope is partially facing the wall (a good thing, as this allows good view of the colon tissue), the lumen would appear on the side or in a corner. In order to maximize the detection of polyps, as much surface area of the

colon should be covered as possible. As the endoscope is withdrawn from the colon, it is often done so in a spiral manner such that the endoscope continues facing the colon wall while rotating, maximizing the surface area seen. While doing this, the lumen will appear to rotate through the various quadrants of the screen.

The development of these modules was crucial in order to study both the efficacy of real-time quality measurements and also the effects of real-time feedback on the quality of a colonoscopy procedure (i.e., would seeing the green triangles in the corner of the screen as feedback as well as a quality score increase the achieved quality of a procedure).

The real-time feedback system has been deployed in two educational rooms at Mayo Clinic. Although SAPPHIRE provides the HUD module with overlay support, certain clinical standards do not allow any delay between the endoscopy machine and the display, so the HUD is not used to display the endoscopy video itself. The purpose of this restriction is because it could be bad, for example, if perceived movement of the endoscope was delayed too much, potentially causing the endoscope to be pushed too far, perhaps perforating the colon. So, instead, we use special overlay hardware that displays the endoscopy video on the monitor as normal. For our HUD, we draw a fullscreen black background and then overlay our feedback video data on top. This signal is fed through a video out channel to the overlay hardware that overlays our video on top of the live colonoscopy procedure video, using the black background as a mask similar to a green screen in video editing techniques. While our feedback may be slightly delayed, the procedure video is displayed instantly behind it. Because of this, there is a small perceived response time for our real-time feedback to be presented back to the physician performing the procedure.

6.13 Summary and Future Work

From a development point of view, SAPPHIRE was used successfully to (1) port an existing program with very little effort, to achieve a higher level of stability and multiprocessing, and (2) create a new application by combining independently developed modules with common data packet formats shared among their programmers. Developers unfamiliar with SAPPHIRE were able to write modules with some effort, although some initial learning curve is expected for any platform. Both of these programs are successfully

deployed in a real hospital, under complex conditions, analyzing actual endoscopy procedures, offering real impact toward patients' quality of health care.

Although we have only presented two related case studies, SAPPHIRE is general enough to be used for any kind of data processing that would benefit from task parallelism (not just medical video). It is especially effective in driving task parallelism in stream programs, increasing the potential of exploitation of parallel processing resources, effectively yielding a faster running program. Although this parallelism is possible without SAPPHIRE or a similar middleware, a great deal more work would be required on the part of the programmer. The API for SAPPHIRE is very simple and it takes much less time to get a new module written and inserted into the program's pipeline.

CHAPTER 7. CONCLUSION AND DISCUSSION OF FUTURE WORK

7.1 Contributions

We have provided contributions in multithreaded software development, theoretical scheduling work, and healthcare:

- SAPPHIRE, including a semi-automated program construction method and multiprocessing framework to enable implicit program multithreading with little effort, allowing simplified exploitation of parallel resources to be used
- A novel static task-scheduling framework for stream programs on a heterogeneous multiprocessor system. Stream-EFT and K-HIT, together provide a novel static stream task scheduling algorithm yielding proveably and simulated near optimal results (for our set of assumptions).
- EM-Capture – We developed novel algorithms and an application for automated real-time endoscopic procedure detection. Our application does not require any human intervention, making it easy to use in a real hospital setting. It has already been used to capture over 71,000 endoscopy procedures. EM-Capture along with SAPPHIRE enables deeper quality analysis programs such as EM-Automated-RT to directly impact the quality of healthcare during important medical procedures.

7.2 Limitations and Future Work

SAPPHIRE is designed with extensions and expansion in mind, so while the design itself is sound, the implementation always has room for expansion through built-in libraries and support for additional platforms. Future work is as follows:

- We plan to continue using SAPPHIRE with our case study programs to examine what additional features may be desireable by real application developers.
- To support devices other than Windows PC's, such as Linux-based servers or handheld devices, the platform-specific code would need to have alternate implementations.

- Future work could augment the SAPPHIRE work loop to support Stream-EFT through user-mode scheduling.
- Stream-EFT can provide optimal scheduling results for our set of assumptions (which may sometimes hold true in practice). But, these assumptions could limit the application of Stream-EFT for other situations. Future work could investigate how to relax some of these constraints.
- We currently expect that computation cost exceeds communication cost (which is true for our case studies); but, this might not be true for other classes of programs. Future work could investigate how to apply Stream-EFT to I/O bound programs.
- We can potentially expand the accuracy and effectiveness of Stream-EFT by adding stochastic scheduling, which more accurately models real life applications with variable task execution times each iteration (as opposed to using static times). Furthurmore, context-based stochastic scheduling, which utilizes statistical contexts based on correlations between tasks' execution times, could provide even more accuracy for estimating the execution time of real programs.
- We can investigate integrating SAPPHIRE with other technologies, such as service oriented computing and cloud computing. This would involve tighter integration with the tcp.dll module, which currently requires explicit specification of computer IP addresses and data types, with manually written configuration files for each system involved. This should be more automated so that the developer has less work. The middleware could provide more efficient load balancing and scheduling than that decided manually by a user.
- We would also like to address security and privacy aspects. For example, if the middleware provides access to sensitive patient information, we might wish to restrict access of that data to specific trusted modules, rather than allow any third party module to access the data.

REFERENCES

[1] S. Stanek et al. Automatic Real-Time Detection of Endoscopic Procedures Using Temporal Features. Computer Methods and Programs in Biomedicine, May 2011.

[2] S. Stanek et al. SAPPHIRE Middleware and Software Development Kit for Medical Video Analysis. Computer-Based Medical Systems, June 2011.

[3] N. Srinivasan et al. A Novel System Able to Provide Real-Time Feedback During Colonoscopy. Abstract. Digestive Disease Week 2012, San Diego, CA, May 2012.

[4] Mark D. Pesce, Programming Microsoft DirectShow for Digital Video and Television, Microsoft Press, 2003.

[5] D. Thomas, P. Moorby. The Verilog Hardware Description Language (5th Edition), Springer, 2002. ISBN 978-0387849300.

[6] Mark Zwolinksi. Digital System Design with VHDL (2nd Edition), Prentice Hall, 2004. ISBN 978-0130399854

[7] G. Kiczales et al. Aspect-Oriented Prorgramming. ECOOP '97. Lecture Notes in Computer Science, pp. 220-242.

[8] R. Filman et al. Aspect-Oriented Software Development. Addison-Wesley Professional, 2004. ISBN 0321219767.

[9] M. Snir et al. MPI: The Complete Reference. MIT Press Cambridge, MA, USA, 1995. ISBN 0262691841.

[10] W. Gropp et al. A high-performance, portable implementation of the MPI message passing interface standard. Parallel Computing, Volume 22, Issue 6, Elsevier, September 1996, pp. 789-828.

[11] NVIDIA, CUDA Zone – The resource for CUDA developers, http://www.nvidia.com/object/cuda_home.html

[12] D. Timothy, R. Hartley, U. Catalyurek, A. Ruiz, F. Igual, R. Mayo, and M. Ujaldon, Biomedical image analysis on a cooperative cluster of GPUs and multicores, Int'l Conf. on Supercomputing, 2008, pp. 15-25.

[13] R. Strzodka, and C. Garbe, Real-Time Motion Estimation and Visualization on Graphics Cards, Conference on Visualization '04, 2004, pp. 545-552.

[14] J.-P. Farrugia, P. Horain, E. Guehenneux, and Y. Alusse, GG-PUCV: A Framework for Image Processing Acceleration with Graphics Processors, IEEE ICME, Toronto, Ontario, CA, 2006, pp. 585-588.

[15] J. Fung, and S. Mann, OpenVidia: Parallel GPU Computer Vision, 2005.

[16] S. Hong and H. Kim. An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness. ICSA'09. pp. 152-163, Austin, Texas, USA, June 2009.

[17] Thies, W., Karczmarek, M., and Amarasinghe, S. P. 2002. StreamIt: A Language for Streaming Applications. In *Proceedings of the 11th international Conference on Compiler Construction* (April 08 - 12, 2002). R. N. Horspool, Ed. Lecture Notes In Computer Science, vol. 2304. Springer-Verlag, London, 179-196.

[18] Khronos Group, OpenCL, http://www.khronos.org/opencl/

[19] J. Stone et al. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. Comput Sci Eng. May 2010; 12(3):66-72.

[20] L. Chen, O. Villa, S. Krishnamoorthy, G. Gao, Dynamic Load Balancing on Single- and Multi-GPU Systems, IPDPS, April 2010.

[21] P. Dutot, T. N'Takpe, F. Suter. Scheduling Parallel Task Graphs on (Almost) Homogeneous Multicluster Platforms. IEEE Transactions on Parallel and Distributed Systems. 20(7), 940-952, July 2009.

[22] M. R. Garey and D. S. Johnson, Computes and Intractability: A guide to the Theory of NP-Completeness.: W. H. Freeman and Company, 1979.

[23] Andrew S. Tanenbaum. Modern Operating Systems (3rd Edition). Prentice Hall, 2007. ISBN 978-0136006633.

[24] Nadathur Rajagopalan Satish. Compile Time Task and Resource Allocation of Concurrent Applications to Multiprocessor Systems. University of California at Berkeley, Technical Report No. UCB/EECS-2009-19. Jan 2009.

[25] T. N'Takpe, F. Suter. Critical Path and Area Based Scheduling of Parallel Task Graphs on Heterogeneous Platforms. In Proc. of Int'l Conf. on Parallel and Distributed Systems (ICPADS'06), 2006.

[26] H. Kasahara and S. Narita. Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing. IEEE Transactions on Computers, C-33(11), Nov. 1984.

[27] Y. K. Kwok and I. Ahmed. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. ACM Computing Survey, 31(4):406-471, 1999.

[28] Task Allocation and Scheduling of Concurrent Applications to Multiprocessor Systems. PhD. Thesis, University of California, Berkeley, Nov. 2007.

[29] G. C. Sih and E. A. Lee. A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Achitectures. IEEE Trans. On Parallel and Distributed System. 4(2):175-187, 1993.

[30] Jie Li, Hisao Kameda. Load Balancing Problems for Multiclass Jobs in Distributed/Parallel Computer Systems. IEEE Trans. On Computers, 47(3):322-332, March 1998.

[31] S. Baskiyar, C. Dickinson. Scheduling Directed A-cyclic Task Graphs on a Bounded Set of Heterogeneous Processors Using Task Duplication, Journal of Parallel Distributed Computing, 2005, pp. 911-921.

[32] T. Adam, K. Chandy, J. Dickson. A Comparison of List Schedules for Parallel Processing Systems. Communications of the ACM, 17(12):685-690, 1974.

[33] Tao Yang, A. Gerasoulis. DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors. IEEE Transactions on Parallel and Distributed Systems. 5(9):951-967, 1994.

[34] E. Hou, N. Ansari, Hong Ren. A Genetic Algorithm for Multiprocessor Scheduling. IEEE Transactions on Parallel and Distributed Systems. 5(2):113-120, 1994.

[35] G. Sih, E. Lee. A Compile Time Scheduling Heuristic for Interconnection Constrained Heterogeneous Processor Architectures. IEEE Transactions on Parallel and Distributed Systems. 4(2):175–187, 1993.

[36] H. Oh, S. Ha. A Static Scheduling Heuristic for Heterogeneous Processors. Lecture Notes in Computer Science, Euro-Par 1996, vol. 1124, pp. 573-577.

[37] H. Topcuoglu, S. Hariri, M. Wu. Task Scheduling Algorithms for Heterogeneous Processors. Heterogeneous Computing Workshop, 1999, pp. 3-14.

[38] Y. Kwok, I. Ahmad. FASTEST: A Practical Low-Complexity Algorithm for Compile-Time Assignment of Parallel Programs to Multiprocessors. IEEE Transactions on Parallel and Distributed Systems. 10(2):147-159, 1999.

[39] A. Radulescu, A. van Gemund. Low-Cost Task Scheduling in Distributed-Memory Machines. IEEE Transactions on Parallel and Distributed Systems. 13(6):648-658, 2002.

[40] A. Radulescu, A. van Gemund. Fast and Effective Task Scheduling in Heterogeneous Systems. IEEE Transactions on Parallel and Distributed Systems. Heterogeneous Computing Workshop, 2000, pp. 229-238.

[41] H. Topcuoglu, S. Hariri, M. Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing Parallel and Distributed Systems. IEEE Transactions on Parallel and Distributed Systems. 13(3):260-274, 2002.

[42] C. Hanen, A. Munier. A Study of the Cyclic Scheduling Problem on Parallel Processors, Discrete Applied Mathematics, Volume 57, Issues 2-3, 24 February 1995, pp. 167-192.

[43] Mark E. Russinovich, David A. Solomon. Microsoft Windows Internals. Microsoft Press, 2005.

[44] American Cancer Society. Colorectal Cancer Facts & Figures. American Cancer Society, 2011.

[45] D. K. Rex, J. L. Petrini, T. H. Baron, A. Chak, J. Cohen, S. E. Deal, B. Hoffman, B. C. Jacobson, K. Mergener, B. Pertersen, M. A. Safdi, D. O. Faigel, and I. M. Pike. Quality indicators for colonoscopy. Gastrointestinal Endoscopy, vol. 63, pp. S16-S26, 2006.

[46] S. Vijan, J. Inadomi, R. A. Hayward, T. P. Hofer, and A. M. Fendrick. Projections of demand and capacity for colonoscopy related to increasing rates of colorectal cancer screening in the United States. Aliment Pharmacol Ther, vol. 20, pp. 507-515, 2004.

[47] Pabby, R. E. Schoen, J.L. Weissfeld, et al. Analysis of colorectal cancer occurrence during surveillance colonoscopy in the dietary Polyp Prevention Trial. Gastrointestinal Endoscopy 2005;61(3):385-91.

[48] L. J. Hixson, M. B. Fennerty, R. E. Sampliner, D. McGee, H. Garewal. Prospective study of the frequency and size distribution of polyps missed by colonoscopy. Journal of the National Cancer Institute 1990;82(22):1769-72.

[49] D. K. Rex, C. S. Cutler, G. T. Lemmel, et al. Colonoscopic miss rates of adenomas determined by back-to-back colonoscopies. Gastroenterology 1997;112(1):24-8.

[50] D. Simmons, G. Harewood, T. Baron, P. Bret, K. Wang, F. Enders, B. Ott. Impact of Endoscopist Withdrawal Speed On Polyp Yield: Implications for Optimal Colonoscopy Withdrawal Time. Gastrointestinal Endoscopy, Volume 63, Issue 5, Pages AB81-AB81.

[51] R. L. Barclay, J. J. Vicari, A. S. Doughty, J. F. Johanson, R. L. Greenlaw. Colonoscopic withdrawal times and adenoma detection during screening colonoscopy. New England Journal of Medicine 2006;355(24):2533-41.

[52] C. Petersohn. Logical unit and scene detection: a comparative survey. (T. Gevers, R. C. Jain, and S. Santini, Editors), Multimedia Content Access: Algorithms and Systems II, Vol. 6820.

[53] M. Coimbra, P. Campos, and J. P. S. Cunha, Topographic segmentation and transit times estimation for endoscopic capsule exams, in Proc. of IEEE Int'l Conf. on Acoustics, Speech, and Signal Processing, Vol. II, pp. 1164-7 (Toulouse, France).

[54] M. Mackiewicz, J. Berens, M. Fisher, Wireless Capsule Endoscopy Colour video segmentation. IEEE Transactions on Medical Imaging; 27 (12):1769-1781 (2008).

[55] J. Lee, J. Oh, S. K. Shah, X. Yuan, and S. J. Tang, Automatic classification of digestive organs in wireless capsule endoscopy videos. In Proc. of ACM Symposium on Applied Computing (Seoul, Korea, 2007).

[56] Karargyris and N. Bourbakis, "A video-frame based registration using segmentation and graph connectivity for Wireless Capsule Endoscopy," Life Science Systems and Applications Workshop, 2009. LiSSA 2009. IEEE/NIH, pp.74-79 (April 2009).

[57] L. Alexandre, N.N. Nobre, and J. C. Casteleiro. Color and Position versus Texture Features for Endoscopic Polyp Detection, Proc. of Int'l Conf. on BioMedical Engineering and Informatics, Vol. 1, pp. 38 – 42 (Sanya, China, May 2008).

[58] D. C. Cheng, W. C. Ting, Y. F. Chen, Q. Pu, and X. Y. Jiang. Colorectal Polyps Detection Using Texture Features and Support Vector Machine, Proc. of Int'l Conf. on Advances in Mass Data Analysis of Images and Signals in Medicine, Biotechnology, Chemistry and Food Industry, pp. 62-72 (2008).

[59] D. K. Iakovidis, D. E. Maroulis, and S. A. Karkanis, An Intelligent System for Automatic Detection of Gastrointestinal Adenomas in Video Endoscopy, Computers in Biology and Medicine, Article in Press (Elsevier Science, 2006).

[60] S. Hwang, J. Oh, W. Tavanapong, J. Wong, and P. C. de Groen, Polyp Detection in Colonoscopy Video Using Elliptical Shape Feature, IEEE Int'l Conf. on Image Processing, pp. 465-468 (San Antonio, TX, USA 2007).

[61] S. Gross, M. Kennel, T. Stehle, J. Wulff, J. Tischedorf, C. Trautwein, and T. Aach, Polyp Segmentation in NBI Colonoscopy (Bildverarbeitung für die Medizin 2009 Springer Link, pp. 252-256. http://www.springerlink.com/content/m7418r5356t13455/).

[62] T. Stehle, R. Auer, S. Gross, A. Behrens, J. Wulfl, T. Aach, R. Winograd, C. Trautwein, and J. Tischendorf, Classification of Colon Polyps in Endoscopy Using Vascularization Features, Proc. of SPIE Medical Imaging, Vol. 7260 (Orlando, USA, Feb 7-12, 2009).

[63] J. Oh, S. Hwang, Y. Cao, W. Tavanapong, J. Wong, and P. C. de Groen. Measuring Objective Quality of Colonoscopy, IEEE Transactions on Biomedical Engineering; 56(9):2190 – 2196 (Sept. 2009).

[64] Y. Wang, W. Tavanapong, J. Wong, J. Oh, and P. C. de Groen, Detection of Quality Visualization of Appendiceal Orifices using Local Edge Cross-Section Profile Features and Near Pause Detection, IEEE Transactions on Biomedial Engineering; 57(3): 689-695 (2010).

[65] D. Hong, W. Tavanapong, J. Wong, J. Oh, and P. C. de Groen, 3D Reconstruction of Colon Segments from Colonoscopy Images, Proc. of IEEE Int'l Conf. on Bioinformatics and Bioengineering, pp. 53-60, (Taiwan, June 2009).

[66] K. Deguchi, Shape Reconstruction from Endoscope Image by its Shadings. IEEE/SICE/RSJ Int'l Conf. on Multisensor Fusion and Integration for Intelligent Systems, Vol. 8. pp. 321-328 (1996).

[67] D. Koppel, C. Chen, Y. Wang, H. Lee, J. Gu, A. Poirson, and R. Wolters, Toward Automated Model Building from Video in Computer-Assisted Diagnoses in Colonoscopy, In Proc. of SPIE Medical Imaging Conference (San Diego, CA, USA, 65091L, 2007).

[68] Kaufman and J. Wang, 3D Surface Reconstruction from Endoscopic Videos, Mathematics and Visualization, Springer Berlin Heidelberg, pp. 61-74 (2007).

[69] G. Bradski and A. Kaebler, Learning OpenCV, Computer Vision with the OpenCV Library (O'Reilly Media, Inc. 2008).

[70] Jochen Kalmbach. Why GetThreadTimes is wrong. http://blog.kalmbachnet.de/?postid=28

[71] N. R. Satish, K. Ravindran, and K. Keutzer. Scheduling Task Dependence Graphs with Variable Task Execution Times onto Heterogeneous Multiprocessors. In Proc. of Int'l Conf. on Embedded Software, Atlanta, Georgia, USA. pp. 149-158, 2008.

[72] Olivier Beaumont, Arnaud Legrand, Loris Marchal, Yves Robert. Assessing the impact and limits of steady-state scheduling for mixed task and data parallelism on heterogeneous platforms. INRIA 5198, 2004.

[73] Kasahara Lab., Waseda University. (2011, September 30). Standard Task Graph Set. http://www.kasahara.elec.waseda.ac.jp/schedule/index.html

[74] S. Stanek, A Soft-Real Time System for Automatic Capture of Colonoscopy Video, MS Thesis, Department of Computer Science, Iowa State University, 2007.

[75] S. Stanek, W. Tavanapong, J. Wong, J. Oh, and P.C.d. Groen, Automatic Real-Time Capture and Segmentation of Endoscopy Video, In Proc. of SPIE Medical Imaging, San Jose, CA, USA, February 2008, pp. 69190X-69190X-10.

[76] M. Sonka, V. Hlavac, and R. Boyle, Image Processing Analysis and Machine Vision, 2nd ed, (Thomson Learning, 1999).

APPENDIX A. SAPPHIRE API

The EndoMetric core API provides a set of functions that modules can call to interact with the middleware. An emCallbacks* object is passed during module initialization to emmRegister containing the function pointers to the middleware. This is usually stored as an object named 'emc', such that modules can interact with the middleware by calling emc->functionname(). More important functions are described in detail, while other functions are discussed only briefly.

A.1 High-level macros

These functions are implemented as short macros to simplify common sequences of events, such as creating a packet. They are defined outside of the 'emc' object and should not prefaced with the 'emc' object (i.e., uses functionname() rather than emc->functionname()). Some macros will assume a naming convention described similar to the naming convention described in the emcGetMeta macro section.

defaultRegister ()

Should be the first statement inside the emmRegister function. This sets up important variables like 'emc' and registers module version information. In the future, if the registration process changes, this macro may be changed so that only a recompile of the module code is necessary, rather than a change in the module code itself.

defaultDllMain ()

Contains a default function body for DllMain, which is called by the operating system upon loading a DLL. This sets up some important variables which may be necessary for interacting with the operating system.

ModuleParameters (char* version, char* description, int internal)

Should be declared somewhere in the module, preferably at the top of the implementation, after including common.h, but before emmRegister is defined. This declares the module version information and a description.

Parameters:

version – a string containing the module version (e.g., "1.0.0")

description – a string containing a description of the module (e.g., "Computes average mean-red value")

internal – a flag denoting whether the module should be hidden and grouped together as a core component (most modules will use 0)

Example:

```
ModuleParameters ( "1.0.0", "End of insertion detection", 0 );
```

emcPreparePacket (packet, meta, metatype, METATYPE, inherit)

Used to create a new packet and setup the metadata structure and all associated packet variables. Specifically, it will create and assign an emPacket* pointer (packet), create and assign a metadata pointer (meta) based on a defined structure (metatype) and string type (METATYPE). The packet and meta variables must already be defined. This macro can be used to setup most types of packets, but should not be used to setup packets that contain extra data attachments like video (using the ->data pointer) or otherwise special packets that are not typically used.

Parameters:

packet – the variable declared as an emPacket* that will receive the newly created packet

meta – the variable declared as a meta_X structure (where X is based on the data type)

metatype – the name of the meta_X structure (e.g., meta_video)

METATYPE – the string name of the data type (e.g., "VIDEO")

inherit – a source packet used to inherit key fields in the packet chain, such as the timestamp

Example:

```
emPacket* eoi;
meta_eoi* metaeoi;
  ...
emcPreparePacket(eoi, metaeoi, meta_eoi, "EOI", pktin);
```

emcGetMeta (name, metatype, mux)

Used to pull the first timestamp's emPacket* and meta_X* pointers out of a mux (if available). The results are stored into variables named according to convention, where the emPacket* variable is named 'name' and the meta_X* variable is declared as a meta_name* structure named 'metaname'. The metatype is the string type of the data packet.

Parameters:

name – the base name of the type that is declared as each of the packet variable, metadata structure name, and metadata variable name

metatype – the string name of the data type (e.g., "VIDEO")

mux – the mux to retrieve the packet from

Example:

```
emPacket* video;
meta_video* metavideo;
  ...
emcGetMeta(video, "VIDEO", locals->mux);  // video and metavideo now filled
```

emcStartClock () / emcStopClock ()

Manually starts and stops the high-precision automated performance gathering system for the current compute thread. Usually not necessary (since these clocks are started and stopped automatically), unless an artificial wait is performed that the middleware does not normally look for. For example, waiting on a mutex does not perform any useful computation, but it does utilize computing resources while waiting. On the other hand, waiting on some other events might not utilize any computing resources, depending on the function used. The usage of these functions is at the discretion of the writer of the module. In general, it is recommended to call emcStopClock() *immediately before* calling a wait function and call emcStartClock() *immediately after* to start the clock back up.

Example:

```
  ...
emcStopClock();
WaitForMultipleObjects(noOfThreads, handles, TRUE, INFINITE);
emcStartClock();
  ...
```

A.2 Core API functions

This section describes the core API of the middleware. Functions are grouped by type. These functions are accessible by calling member functions of the 'emc' object, which is an internal object setup in defaultRegister(). These functions will not be accessible until the 'emc' object has been properly setup. Unlike the macros, these functions are called by prefacing with the 'emc' object (e.g., emc->emcAddInput(...)). Nearly all functions require the first parameter of 'module' to be specified, which is simply the module passed in through emmRegister, emmData, etc.

A.2.1 Registration

Module registration typically consists of configuring data inputs and outputs. User configuration information is passed to the module's emmRegister function so that the module can properly indicate which inputs and outputs it is willing to accept or generate. Single inputs are added with emcAddInput(), while outputs are added with emcAddOutput(). Complex inputs consisting of multiple input data types can be grouped together into a single logical input by using a mux (multiplexer). This higher level construct helps by providing simple synchronization functions to make working with multiple data types easier. Data types are first added to a mux with emcAddMux(), and then the mux is registered by calling emcAddInputMux().

Because synchronization is not guaranteed with data types added with emcAddInput(), emcAddInputMux() is preferable in almost all cases. Even if using just a single input data type, the mux framework will work just as efficiently (negligible overhead). The advantages of using a mux instead of the low-level emcAddInput(), even for just a single packet type, is

that the middleware provides higher level functions that are useful for packet handling, and it is much easier to add additional inputs with synchronization later, rather than changing a significant amount of code to handle the change.

int emcAddInput (emModule* module, char* type, int* meta, int metalen)

Simple inputs are added with this function. The type of the data input is specified by the string 'type'. Before using this function, please consider the note above about how emcAddInputMux() is almost always preferred over this function. The 'meta' and 'metalen' parameters optionally assign a metadata filter to the input; however, this functionality is currently not very useful, so it is only partially implemented into the framework. Generally, a default value of 0 should be specified for 'meta' and 'metalen'.

Parameters:

type – the string name of the data type (e.g., "VIDEO")

meta (optional) – pointer to a metadata filter structure if desired

metalen (optional) – length of metadata structure

Example:

```
emc->emcAddInput(module, "VIDEO", 0, 0);  // request VIDEO data type
```

int emcAddOutput (emModule* module, char* type, int* meta, int metalen, int maxchainlen)

Simple outputs are added with this function. The type of the data to be output is specified by the string 'type'. There is no 'mux' version associated with outputs since synchronization for muxes are handled by the middleware and by the input functions. Metadata filters can be specified, although this is a rarely used feature. Support for this feature may not be fully implemented; a default value of 0 is usually specified. The maximum packet chain length for a particular output type can be overridden with the 'maxchainlen' parameter. This is the number of packets of a particular output type that the system will buffer before considering that buffer full, and begin blocking requested outputs until free space is available. For large packets such as video, this value is usually specified to minimize the maximum memory usage. For small packets, this parameter can usually be ignored. The default of 3000 packets is used if 0 is specified.

Parameters:

type – the string name of the data type (e.g., "MEANR")

meta (optional) – pointer to a metadata filter structure if desired

metalen (optional) – length of metadata structure

maxchainlen (optional) – maximum packet chain length

Example:

```
emc->emcAddOutput(module, "MEANR", 0, 0, 0);  // MEANR data type will be
                                                 output
```

int emcAddMux (emModule* module, emMux* mux, char* type, int* meta, int metalen, int optional, int delay)

Adds a specified data type into an existing mux. The mux is currently created by the module programmer (e.g., mux = calloc(...)) rather than by the middleware, although this may change in a future version. The data types are not actually registered by adding them to the mux alone; this must be followed up by registering the mux with the middleware with emcAddInputMux(). The 'optional' flag specifies whether an input is optional or not. If false (zero), the data type must exist in order for the mux to be satisfied. If true (non-zero), the mux can be satisfied regardless of the existence of the specified data type. If 'optional' is true and the data type is registered as an output in the system, mux synchronization will correctly wait for this data type (i.e., the data type's existence itself is optional, rather than a packet of that existing data type being available).

In some circumstances, a feedback loop for data may be desired. Because it is impossible to receive output data for a particular timestamp from another module that does not yet have that timestamp's input data (a circular dependence), a delay can be specified to allow the mux to receive a previous timestamp's data packet. The 'delay' parameter is, optionally, a positive number specifying such a delay. The default of 0 specifies that no such delay exists for that particular data type.

Note that the first 'delay' number of packets for a satisfied mux for this data type will consist of a null packet. For 'optional' inputs, the same is true (a null packet being returned upon requesting a particular data type). Module programmers should take care to check optional and delay type packets to make sure they are non-null before using them.

Parameters:

mux – an existing mux to add this input data type to

type – the string name of the data type (e.g., "VIDEO")

meta (optional) – pointer to a metadata filter structure if desired

metalen (optional) – length of metadata structure

optional (optional) – true/false flag to specify whether the data type is optional

delay (optional) – specify an amount of delay (in time quantums) for this data type

Example:

```
emMux* mux;  // note: declared inside a structure named 'locals'
  ...
emc->emcAddMux(module, locals->mux, "VIDEO", 0, 0, 0, 0);
```

int emcAddInputMux (emModule* module, emMux* mux)

Adds this mux as an input for the module. This function takes all the inputs previously added to the mux with emcAddMux() and then actually adds each of them to the module itself. No more inputs should be added to the mux after this function is called.

Parameters:

mux – an existing mux with input data types to register with the module

Example:

```
emc->emcAddInputMux(module, locals->mux);  // register this mux with the
                                                module
```

Rarely used registration functions

The following functions are not normally used. The module version registration functions are already called by defaultRegister(), and thus, they need not be called manually. However, they are provided below for reference. The emcSetPacketFilter is not called by any macros, but it is also not normally used – it is mainly used to set filter points on data types, similar to hooking functions or aspect-oriented bindings.

int emcSetPacketFilter (emModule* module, char* type, int* meta, int metalen, unsigned int priority)

Assigns a filter to a particular data type using a specific priority.

Parameters:

type – the string name of the data type (e.g., "MEANR")

meta (optional) – pointer to a metadata filter structure if desired

metalen (optional) – length of metadata structure

priority – the priority of the packet filter (higher indicates an earlier location in the filter chain)

Example:

```
emc->emcAddInput(module, "VIDEO", ... );
emc->emcAddOutput(module, "VIDEO", ... );
emc->emcSetPacketFilter(module, "VIDEO", 0, 0, 1000);
```

int emcRegisterModuleVersion (emModule* module, char* versionstring, int buildnumber, char* builddate)

Registers the module version information. Usually this function is called through defaultRegister() rather than directly.

Parameters:

versionstring – the version string of the SDK

buildnumber – the SDK build number this module was built with

builddate – the date and timestamp of the module compile

Example:

```
emc->emcRegisterModuleVersion(module, EM_VERSION, EM_BUILDVER, __DATE__ ##
                                                          " " ## __TIME__);
```

int emcRegisterModuleVersion2 (emModule* module, char* versionstring, char* description, int internal)

Alternative method to registering module version information.

Parameters:

verionstring – the version string of the module

description – a textual description of the module's function

internal – f lag to indicate whether or not this module is a built-in module

Example:

```
emc->emcRegisterModuleVersion2(module, EM_MODULEVERSION,
                                  EM_MODULEDESCRIPTION, EM_MODULEINTERNAL);
```

int emcSetDataTypeOffset (emModule* module, char* type, char* varname, int offset, char* vartype, int varsize, int numvar)

Registers a metadata structure variable with a friendly string name so that other modules and programmers can access structure members by name at runtime instead of by binary compilation with the structure format (the header file for the structure). Although binary compilation with the structure format yields faster code, referencing by variable name may be more friendly in some circumstances (e.g., displaying packet variable values in hud.dll by specifying the variable names in the configuration file, rather than having to specify their byte offsets). The code to register a full structure can be automatically generated by the gendataoffset.exe program included as a tool with the SDK.

Parameters:

type – the string name of the data type (e.g., "VIDEO")

varname – a string of the C type of the structure variable (e.g., "int")

offset – the byte offset of the structure variable (e.g., 4)

vartype – a string representing the type of the variable (e.g., "integer")

varsize – the size in bytes of the variable type (e.g., 4)

numvar – for an array, the number elements in the array

Example:

```
emc->emcSetDataTypeOffset(module, "VIDEO", "height",
   (int)((char*)&__tmp_meta_video.height - (char*)&__tmp_meta_video), "int",
   sizeof(int), sizeof(int)/sizeof(int));
```

typeinfo* (*emcGetDataTypeOffset) (emModule* module, char* type, char* varname)

Retrieves the type info for a type and structure variable name set by emcSetDataTypeOffset. Note that this is not actually a registration function, but rather, this is the counterpart to emcSetDataTypeOffset function used in registration. It will not be valid until all modules have registered and emmStart has been called.

Parameters:

type – the string name of the data type to retrieve information about (e.g., "VIDEO")

varname – the string name of the structure variable to retrieve information about (e.g. "height")

Example:

```
ti = emc->emcGetDataTypeOffset(module, "VIDEO", "height");
```

A.2.2 Packets

Packet functions are typically used to create, modify, and release data in the system. They are the primary method of achieving communication between modules.

emPacket* emcNewPacket (emModule* module, int metalen, int datalen)

Creates a new packet, to be filled with data by the programmer. This packet must be pushed to the system or it will become a memory leak.

Parameters:

metalen – the size of the data required by the metadata structure ->meta

datalen – the size of the data required by the attached data structure ->data

Example:

```
pkt = emc->emcNewPacket(module, sizeof(meta_raw_video), width * height * 4);
  ...
emc->emcPushPacket(module, pkt);
```

int emcPushPacket (emModule* module, emPacket* pkt)

Pushes the packet of data to the middleware. All fields must be finished before this function is called. Memory that was allocated inside this packet may become invalid immediately after pushing (as it could become used and freed quickly), so modules should ensure that any data that should persist after calling emcPushPacket should be copied somewhere else or allocated outside of normal packet deallocation flow.

Parameters:

pkt – an allocated, preformatted, filled out packet structure to be pushed to the middleware

Example:

```
pkt = emc->emcNewPacket(module, sizeof(meta_meanr), 0);
  ...
emc->emcPushPacket(module, pkt);
```

int emcMuxPacket (emModule* module, emMux* mux, emPacket* pkt, int num)

Adds a packet of data to a mux. If the mux is satisfied (has at least *num* of each packet available), then this function returns *num*. Otherwise, it returns 0.

Parameters:

mux – the mux structure to insert the packet into

pkt – the packet passed in through emmData

num – the requested number of packets that must be inserted into the mux across all data types

Example:

```
// this will just exit an emmData function immediately
// if the mux is not satisfied
```

```
if(!emc->emcMuxPacket(module, mux, pkt, 1)) return 0;
```

int emcReleaseMuxPackets (emModule* module, emMux* mux, int num)

Releases *num* packets of each data type listed in the mux. This removes the packets both from the mux and releases them from the module.

Parameters:

mux – the mux structure to release packets from

num – the number of packets of each type to release from the module

Example:

```
if(!emc->emcMuxPacket(module, mux, pkt, 1)) return 0;
  ... process data ...
emc->emcReleaseMuxPackets(module, mux, 1);
```

emPacket* emcFindPacket (emModule* module, emMux* mux, char* type)

Retrieves a packet of a specific type from the mux. If no packet of that type is available, this function returns 0.

Parameters:

mux – the input mux to retrieve a packet from

type – the string name of the data type (e.g., "MEANR")

Example:

```
if(!emc->emcMuxPacket(module, mux, pkt, 1)) return 0;
meanr = emc->emcFindPacket(module, mux, "MEANR");
```

int emcPushDummyPacket (emModule* module, char* type, __int64 timestamp, int final)

Pushes a dummy packet of a specific type, synchronized with a specific timestamp and final flag. The timestamp and final flag are usually passed in through some parent packet. This function manages the full creation and pushing of a packet, so no other calls to emcNewPacket or emcPushPacket are needed for the dummy packet.

Parameters:

type – the string name of the data type (e.g., "MEANR")

timestamp – the timestamp to use to push a dummy packet

final – a flag denoting whether or not this will be the final packet for this data stream

Example:

```
if(skipframe) {
  emc->emcPushDummyPacket(module, "MEANR", pktin->timestamp, pktin->final);
} else {
  pkt = emc->emcNewPacket(...);
    ...
  emc->emcPushPacket(...);
}
```

int emcReleasePackets (emModule* module, int j, int r)

Releases packets of internal packet index type *j* and number of packets *r*. This is less used due to the flexibility of the mux functionality and the fact that returning a positive number from emmData automatically generates a call to this function.

Parameters:

j – module's internal packet type index of which to release a packet

r – number of packets to release

Example:

```
emc->emcReleasePackets(module, j, r);
```

int emcAddReleaseCallback (emModule* module, emPacket* pkt, int (*callback)(emPacket* pkt))

Adds a callback function to be called upon just prior to a packet's actual deallocation. This allows a programmer to attach dynamic memory and other objects to a packet and allows those objects to be properly freed. The freeing of the packet data itself is handled by the middleware; the module creating a packet need not free the packet, only extra pointer data attached to it that the middleware would not understand how to deallocate itself.

Parameters:

pkt – the packet to attach a custom packet destructor callback function to

callback – the callback function to call

Example:

```
int freeColonCenter(emPacket* pkt) { delete pkt->cc; }
  ...
spiralpkt = emc->emcNewPacket(...);
spiralpkt->cc = new ColonCenter();
emc->emcAddReleaseCallback(module, spiralpkt, freeColonCenter);
emc->emcPushPacket(module, spiralpkt);
```

int emcInheritPacket (emPacket* packet, emPacket* inherit)

Inherits several fields from a parent packet *inherit*, such as the timestamp and final flag, and copies them to *packet*.

Parameters:

packet – destination packet to copy values to

inherit – source packet to inherit values from

Example:

```
pktout = emc->emcNewPacket(...);
emc->emcInheritPacket(pktout, pktin);
  ... fill pktout ...
emc->emcPushPacket(module, pktout);
```

emPacket* emcCopyPacket (emModule* module, emPacket* pkt)

Performs a deep copy of a packet and returns the copy.

Parameters:

pkt – packet to copy

Example:

```
newpkt = emc->emcCopyPacket(module, pktin);
```

emPacket* emcCopyPacketShallow (emModule* module, emPacket* pkt)

Performs a shallow copy of a packet and returns the copy (data pointers will point to the parent packet).

Parameters:

pkt – packet to copy

Example:

```
newpkt = emc->emcCopyPacketShallow(module, pktin);
```

int emcDataTypeExists (emModule* module, char* type)

Returns whether or not a data type exists in the overall system. This will not be valid until after all modules have registered (i.e., this function cannot be called during emmRegister, and should not be called until emmStart or later).

Parameters:

type – the string name of the data type (e.g., "MEANR")

Example:

```
if(emc->emcDataTypeExists(module, "INSIDE")) {
  ... modify module behavior due to existence of "INSIDE" data ...
}
```

A.2.3 Control

int emcSetFinished (emModule* module, int status)

Sets a module to a finished state. The middleware automatically determines when a module has entered a finished state based on the completion of processing of all incoming packets as well as the finished states of all modules producing those packets. This function is only necessary for source type modules (e.g., mpegreader.dll) for which the middleware has no implicit way to determine the completion of the module.

Parameters:

status – whether this module is finished (nonzero) or not (zero)

Example:

```
if(done) emc->emcSetFinished(module, 1);
```

A.2.4 Performance

double emcGetPushTime (emModule* module, __int64 pushtime)

Returns the number of seconds since SAPPHIRE started, corresponding to an arbitrary reference clock, of a value specified by *pushtime*. This is usually a packet's ->pushtime member variable.

Parameters:

pushtime – number of clock cycles elapsed since SAPPHIRE started

Example:

```
packettime = emc->emcGetPushTime(module, pktin->pushtime);
printf("time since last packet = %.3f ms\n",
                    (packettime – lastpackettime)*1000.0);
lastpackettime = packettime;
```

int emcAddThread (emModule* module, HANDLE hthread, int threadid)

Manually adds a thread to belong to *module*. New threads are almost always automatically detected and manual addition through this function is not needed.

Parameters:

hthread – Windows handle to the created thread

threadid – Windows globally unique thread id of the created thread

Example:

```
HANDLE hthread = CreateThread(..., &threadid);
emc->emcAddThread(module, hthread, threadid);
```

int emcPerfStartClock (emModule* module, char* filename, int line)

The explicit function that the emcStartClock macro calls.

Parameters:

filename – file name this function is in (e.g., __FILE__)

line – line this function is being called from (e.g., __LINE__)

Example:

```
emc->emcPerfStopClock(module, __FILE__, __LINE__);
WaitForSingleObject(...);
emc->emcPerfStartClock(module, __FILE__, __LINE__);
```

int emcPerfStopClock (emModule* module, char* filename, int line)

The explicit function that the emcStopClock macro calls.

Parameters:

filename – file name this function is in (e.g., __FILE__)

line – line this function is being called from (e.g., __LINE__)

Example:

```
emc->emcPerfStopClock(module, __FILE__, __LINE__);
WaitForSingleObject(...);
emc->emcPerfStartClock(module, __FILE__, __LINE__);
```

A.2.5 Miscellaneous

int emcGetModuleInfo (int index, emModule* resultmodule)

Gets the module info for a module specified by *index* and returns the module into a user-provided emModule structure. If *resultmodule* is not specified, this function returns the number of modules in the system. This function is not valid until all modules have registered (i.e., it cannot be used until emmStart is called). Most information about a module can be retrieved through this function, however state-specific information that may quickly expire (such as pointers to a module's next incoming packets) is not retrieved.

Parameters:

index – module number to retrieve info about

resultmodule – destination pointer to an emModule structure to receive information

Example:

```
n = emc->emcGetModuleInfo(0, 0);
for(i=0;i<n;i++) {
    emc->emcGetModuleInfo(i, &m);
    printf("module %2d : %s %s.%d\n", i,m.dllname, m.versionstring,
                m.buildnumber);
}
```

int emcGetTypeNum (char* typestr)
Returns the internal type index for a specified string-formatted data type.

Parameters:

typestr – the string name of the data type (e.g., "MEANR")

Example:

```
typeindex = emc->emcGetTypeNum("MEANR");
```

char* emcGetTypeName (int typenum)
Returns the string-formatted data type name for a given type index.

Parameters:

type – the string name of the data type (e.g., "MEANR")

Example:

```
  // note: this code returns "MEANR" back, but it becomes a
  // pointer to the middleware's internal string of "MEANR"
printf("%s", emc->emcGetTypeName(emc->emcGetTypeNum("MEANR")));
```

void emcDumpPacketChain ()
Prints out the full linked lists of packets currently in the system.

Example:

```
if(inconsistency_found) emc->emcDumpPacketChain();
```

APPENDIX B. MODULES AND FUNCTIONALITY

Modules included with SAPPHIRE	
Module name	**Task performed by the module**
mpegreader.dll	Read MPEG-2 video files
mpegwriter.dll and mpegwriter2.dll	Write MPEG-2 video files
videocapture.dll	Obtain video signal from a video capturing card
hud.dll	Display videos, feedback, and process some user keyboard input
videomixer.dll	Combine multiple VIDEO data types and output a single VIDEO data type that can be used by modules that accept VIDEO data type (e.g., hud.dll or mpegwriter.dll)
screencapture.dll	Capture video from the computer desktop or specified window and output it as a VIDEO stream
log.dll	Synchronized logging support with typical logging levels and options built-in
emlive.dll	Export VIDEO data to external programs in real-time through a system shared memory region
imagemask.dll	Derive image mask from a VIDEO stream and output a new video stream of the image mask
autoresize.dll	Automatically crop/resize a VIDEO stream
grayscale.dll	Generate a grayscale video stream from an input VIDEO stream
tcp.dll	Allow SAPPHIRE to communicate data over network using TCP, enabling cluster and grid computing.

Third-party modules	
Module name	**Task performed by the module**
blurry.dll	Detect informative frames and compute related metrics
egd.dll	Detect whether the video is colonoscopy (default) or upper endoscopy
rts.dll	Detect stool pixels and calculate stool related metrics
rteoi.dll	Detect end-of-insertion frame in real-time
rteoi_user.dll	Accept user-specified end of insertion frame number via keyboard input
spiralCounting.dll	Compute spiral related metrics
ecspEdgeTracking.dll, DetectionECSP_module.dll	Detect potential polyp edges and provide feedback
QCMetricRT.dll	Generate a CSV file with quality measurements for each video analyzed
inside.dll, meanr.dll, meannormr.dll, histdiff.dll, meanr_dv80.dll, color_energy.dll, meannormr_risefall.dll,brightness.dll	Detect the start and end frame of an endoscopic procedure
eoi_eop_gt.dll	Read end-of-insertion and end-of-procedure frame numbers from a CSV file
usbname.dll	Read an encrypted endoscopist's name from a thumb drive.

APPENDIX C. MODULES AND THEIR PACKET TYPES

Modules included with SAPPHIRE		
Module name	**Input packet types**	**Output packet types**
mpegreader.dll		RAW_VIDEO, LOG
mpegwriter.dll	VIDEO, INSIDE	FEEDBACK_VIDEO_FILE_INFO, MOTION_VECTORS
mpegwriter2.dll	VIDEO_MIXER, INSIDE	
videocapture.dll		RAW_VIDEO, LOG
hud.dll	VIDEO, STOOL, BLURRY, QCMETRIC_USER, INSIDE_NO_DELAY, INSIDE, VIDEO_MIXER_OVERLAY, RAW_VIDEO, EOI, EOI_USER	USER_INPUT, HUD_WINDOW_OUTPUT
videomixer.dll	SPIRAL_OVERLAY, ECSP_OVERLAY	VIDEO_MIXER_OVERLAY
screencapture.dll		VIDEO
log.dll	LOG	NULL
emlive.dll	VIDEO	
imagemask.dll	VIDEO, INSIDE	IMAGEMASK
autoresize.dll	RAW_VIDEO, INSIDE_NODELAY	VIDEO
grayscale.dll	VIDEO	GRAYSCALE_VIDEO
tcp.dll	(configurable; data to send over network)	(configurable; data to receive from network)

Third-party modules		
Module name	**Input packet types**	**Output packet types**
meanr.dll	RAW_VIDEO	MEANR
histdiff.dll	RAW_VIDEO	HISTDIFF
meanr_dv80.dll	MEANR	MEANR_DV80
color_energy.dll	MEANR, MEANNORMR	MEAN_ENERGY, MEANNORM_ENERGY, COLOR_ENERGY, MEAN_HIST_CONT_AREA
meannormr_risefall.dll	MEANR, MEANNORMR, MEANR_DV80, INSIDE_NODELAY	MEANNORMR_RISEFALL
brightness.dll	RAW_VIDEO	BRIGHTNESS
inside.dll	MEANR, MEANNORMR, HISTDIFF, MEANR_DV80, MEANNORMR_RISEFALL, MEAN_ENERGY, MEANNORM_ENERGY, COLOR_ENERGY, MEAN_HIST_CONT_AREA, BRIGHTNESS, STOOL, RETRO	INSIDE, INSIDE_NODELAY
rteoi.dll	VIDEO, INSIDE, FEEDBACK_VIDEO_FILE_INFO	LOG, CDCM, EOI
EGD.dll	VIDEO, INSIDE, FEEDBACK_VIDEO_FILE_INFO	EGD
RTS.dll	VIDEO, INSIDE, FEEDBACK_VIDEO_FILE_INFO	STOOL, LOG
blurry.dll	VIDEO, INSIDE, FEEDBACK_VIDEO_FILE_INFO	BLURRY, LOG
spiralcounting.dll	VIDEO, INSIDE, FEEDBACK_VIDEO_FILE_INFO, BLURRY, EOI_USER, EOI	LOG, SPIRAL, SPIRAL_OVERLAY
qcmetricrt.dll	BLURRY, STOOL, INSIDE, FEEDBACK_VIDEO_FILE_INFO, VIDEO, SPIRAL, EGD, EOI_USER, EOI	QCMETRICRT_USER, LOG
rteoi_user.dll	VIDEO, FEEDBACK_VIDEO_FILE_INFO, INSIDE	EOI_USER, LOG
eoi_eop_gt.dll	EOI_USER, VIDEO, INSIDE, RAW_VIDEO, INSIDE_NODELAY	INSIDE_NODELAY, INSIDE, EOI_USER, LOG
ecspEdgeTracking.dll	VIDEO, SPIRAL, IMAGEMASK, FEEDBACK_VIDEO_FILE_INFO	LOG, EDGE, EDGE_OVERLAY, ECSPEDGE
DetectionECSP_Module.dll	VIDEO, INSIDE, BLURRY, SPIRAL, IMAGEMASK, FEEDBACK_VIDEO_FILE_INFO, ECSPEDGE	LOG, DETECTION_ECSP, ECSP_OVERLAY

APPENDIX D. SAMPLE ENDOCAPTURE.INI

D.1 Endoscopic procedure detection and capturing (EM-Capture)

```
[videocapture.dll]
video.device=
video.device.width=720
video.device.height=480
video.mux=Composite
video.width=720
video.height=480
video.bpp=32
video.fps=29.97
video.bufferlen=30
video.tvformat=1                    # 1=NTSC_M, 16=PAL_B

[autoresize.dll]

[meanr.dll]
[meannormr.dll]
[histdiff.dll]

[meanr_dv80.dll]
windowsize = 240

[meannormr_dv80.dll]
windowsize = 240

[color_energy.dll]
windowsize = 479
windowsize_big = 1800

[meannormr_risefall.dll]
windowsize = 240

[inside.dll]
quiet = 1
#recordall = 1
#blackframes = 60
vis=0
threshold.meanR = 8
threshold.meannormR = 33
threshold.meannormR_runningtotal = 2400
threshold.maxstartlength = 240
threshold.outside_duration = 9000
threshold.outside_percent = 0.90
threshold.outside_nosignal = 2700
threshold.motion = 5000
threshold.meanR_dv80 = 2
```

46	threshold.meanR_dv80_end = 0.05
47	threshold.meannormR_dv80 = 0.2
48	threshold.meanR_dv80_2 = 0.2
49	threshold.meannormR_dv80_2 = 0.05
50	threshold.greenarea = 4000
51	threshold.greenarea_low = 100
52	threshold.redarea = 3000
53	threshold.meanR_energy = 64
54	threshold.meannormR_energy = 16
55	threshold.count_energy = 32
56	threshold.count_energy_end = 3.2
57	threshold.count_area = 15000
58	
59	[mpegwriter.dll]
60	quality = 31
61	realtime = 0
62	bitrate = 8000000
63	
64	file = out\%YYYY%%MM%%DD%_%hh%%mm%%ss%_%ip%_P%num%.mpg
65	outside = 120
66	minlength = 600
67	
68	[emlive.dll]
69	
70	[log.dll]
71	logfile = out/log.txt
72	level = 9
73	
74	[brightness.dll]
75	[grayscale.dll]

D.2 Real-time feedback (EM-Automated-RT)

1	[videocapture.dll]
2	video.device.width=720
3	video.device.height=480
4	video.mux=Composite
5	video.width=720
6	video.height=480
7	video.bpp=32
8	video.fps=29.97
9	video.bufferlen=90
10	video.tvformat=1 # 1=NTSC_M, 16=PAL_B
11	
12	[autoresize.dll]
13	
14	[meanr.dll]
15	[meannormr.dll]
16	[histdiff.dll]
17	
18	[meanr_dv80.dll]
19	windowsize = 240
20	
21	[meannormr_dv80.dll]
22	windowsize = 240
23	
24	[color_energy.dll]
25	windowsize = 479
26	windowsize_big = 1800
27	
28	[meannormr_risefall.dll]
29	windowsize = 240
30	
31	[inside.dll]
32	quiet = 1
33	recordall = 0
34	#blackframes = 20
35	vis=0
36	threshold.meanR = 8
37	threshold.meannormR = 33
38	threshold.meannormR_runningtotal = 2400
39	threshold.maxstartlength = 240
40	threshold.outside_duration = 9000
41	threshold.outside_percent = 0.90
42	threshold.outside_nosignal = 2700
43	threshold.motion = 5000
44	threshold.meanR_dv80 = 2
45	threshold.meanR_dv80_end = 0.05
46	threshold.meannormR_dv80 = 0.2
47	threshold.meanR_dv80_2 = 0.2
48	threshold.meannormR_dv80_2 = 0.05

49	threshold.greenarea = 4000
50	threshold.greenarea_low = 100
51	threshold.redarea = 3000
52	threshold.meanR_energy = 64
53	threshold.meannormR_energy = 16
54	threshold.count_energy = 32
55	threshold.count_energy_end = 3.2
56	threshold.count_area = 15000
57	
58	[mpegwriter.dll]
59	quality = 31
60	realtime = 0
61	bitrate = 8000000
62	#file = out\%YYYY%%MM%%DD%_%hh%%mm%%ss%_%ip%_P%num%.mpg
63	file=null
64	#outside = 120
65	#minlength = 600
66	outside = 0
67	minlength = 600
68	
69	[mpegwriter2.dll]
70	#comment out for recording without feedback
71	quality = 31
72	realtime = 0
73	bitrate = 8000000
74	file = out\%YYYY%%MM%%DD%_%hh%%mm%%ss%_%ip%_FB_P%num%.mpg
75	#file = out\test.mpg
76	#outside = 120
77	#minlength = 600
78	outside = 0
79	minlength = 600
80	single=1
81	input=VIDEO_MIXER
82	
83	[hud.dll]
84	hideuntileoi=1
85	#synchronized displayed frames and processing results
86	sync =1
87	statistics=0
88	input=VIDEO
89	#input=SPIRAL_OVERLAY
90	font=Times New Roman:14
91	color=ffffffff # green ff00ff00
92	top-left
93	#blank line
94	##text=QCMETRICRT:168.int/Comp
95	#spiral using computed EOI
96	##text=QCMETRICRT:172.int/S (W):%d
97	##text=QCMETRICRT:168.int/S (I):%d
98	#computed EOI
99	##text=QCMETRICRT:180.int/EOI:%d

100	#withdrawal time based on computed EOI
101	##text=QCMETRICRT:0.string/IT:%9s
102	##text=QCMETRICRT:10.string/WT:%9s
103	#clear withdrawal time based on computed EOI
104	##text=QCMETRICRT:20.string/CWT:%9s
105	##text=QCMETRICRT:60.float/Unclean (%%F) :%.2f
106	##text=SPIRAL:12.int/spiral:%d
107	
108	top-right
109	#text=QCMETRICRT_USER:168.int/User
110	#spiral count using user specified EOI
111	text=QCMETRICRT_USER:172.int/
112	text=QCMETRICRT_USER:172.int/S (W):%d
113	#text=QCMETRICRT_USER:168.int/S (I):%d
114	#user specified EOI
115	text=QCMETRICRT_USER:180.int/%d
116	# EOI removed number shown: text=QCMETRICRT:180.int/%d
117	#text=QCMETRICRT_USER:188.int/isEGD:%d
118	#withddrawal time based on user specified eoi
119	#text=QCMETRICRT_USER:0.string/IT:%9s
120	#text=QCMETRICRT_USER:10.string/WT :%9s
121	#clear withddrawal time based on user specified eoi
122	#text=QCMETRICRT_USER:20.string/CWT :%9s
123	
124	#text=QCMETRICRT_USER:60.float/Dirty (%%F):%.2f
125	#text=QCMETRICRT_USER:68.float/Dirty (%%CF):%.2f
126	
127	overlay=SPIRAL_OVERLAY
128	#overlaycolor=7fffffff
129	#overlaykey=ffff0000
130	
131	useblackvideo=1
132	#used to show only feedback without video signal
133	
134	record=2
135	#record=0 for recording without feedback
136	#record=2 for recording video without the Windows window frame
137	#records video and feedback
138	
139	[log.dll]
140	logfile = out/log.txt
141	level = 1
142	
143	[brightness.dll]
144	[grayscale.dll]
145	
146	[RTS.dll]
147	#frames/second
148	frameAnalysisRate=1
149	
150	[blurry.dll]

151	# frames/second
152	frameAnalysisRate=1
153	
154	[spiralCounting.dll]
155	#number of frames to skip
156	frameAnalysisRate = 1 # default 3.0 frames per second for old version9
157	threshold.laterality = 0.4 # default 0.6
158	threshold.outBound = 0.95 # default 0.9
159	show.lumen=0 # show the detected lumen (do NOT show by default); set it 1 to show; set it 0 not to show
160	show.circles=0 # show both circles (show by default); set it 1 to show; set it 0 not to show
161	show.corners=1 # show a green triangle at the corner when that quadrant has been inspected (show by default); set it 1 to show; set it 0 not to show
162	show.inspecting=0 # mark currently inspecting mucosa area (show by default); set it 1 to show; set it 0 not to show
163	show.corners.size=4 #1 for smallest; 10 for biggest (default 5)
164	#input.eoi=1 #0 for system eoi; 1 for user input eoi
165	input.eoi=1
166	
167	[QCMetricRT.dll]
168	outputfile=out/metric.csv
169	eoi=0
170	eoiuser=1
171	
172	[EGD.dll]
173	frameAnalysisRate=3
174	
175	[rteoi_user.dll]
176	
177	#[rteoi.dll]
178	macroblocksize=16 # Size of a macro block (used in the searching algorithm)
179	searchareasize=8 # Size of the search area (used in the searching algorithm)
180	frameAnalysisRate=3 # Number of frames(pairs) processed per second
181	blockSkip=1 # Number of blocks skipped (1 = no skip, 2 = 1 skip and so on)
182	
183	[videomixer.dll]
184	input=VIDEO
185	input=HUD_WINDOW_OUTPUT
186	output=VIDEO_MIXER

APPENDIX E. MEMORY LEAK DETECTION USAGE INFORMATION

E.1 Instructions

(0) (Optional) Compile your module in debug mode if you wish to trace back potential leaks to your code. This is much easier to do now than in the previous version.

(1) Run: cap -memleak

(2) Wait about 60 seconds for various module initializations to take place, so that these initializations that never get freed are not seen as memory leaks.

(3) Press and hold Ctrl+Shift+Alt+F to open the performance window. This will do the first time initialization of the memory state of the program.

(4) (Optional) Repeat the key combination Ctrl+Shift+Alt+F to close the performance window.

(5) Wait several minutes (or longer). You should change your console window to be larger so that you can properly see all of the memory leaks without them scrolling off. Go to the console window properties (in Windows 7 you may need to left click the top-left icon of the window title bar). Go to the Layout tab. Change the screen buffer size and window size. It is recommended for at least 120 for both widths and 9999 for the screen buffer size height. This will allow you to scroll back much farther than the default.

(6) Close the performance window if it isn't already closed.

(7) Press and hold Ctrl+Shift+Alt+M (M for memory) for about 2-3 seconds. This will reset the "previously displayed leaks" list and cause all the previously displayed possible leaks to be reused (if they are still allocated) when you reopen the performance window.

(8) Press and hold Ctrl+Shift+Alt+F to open the performance window. This will cause all the memory leak candidates from first initialization to be displayed.

E.2 Interpreting the output

Because this implementation is a very strange way to detect memory leaks (without requiring a recompile or special binaries), there is some chance for misdetection. Age and repetition are important factors. There is a 20 second minimum age to print a memory leak canditate. The older it is, the more likely that some piece of data was completely forgotten about. If an address is only seen once over a 20 minute gathering period, it may be a fluke. But, if an address is seen 30 times per second for several minutes, there would be a strong possibility that there is a memory leak every frame.

E.3 Example output and debugging

```
| [        spiralCounting.dll+00004ea0 = 06674ea0 ] allocated    131064 bytes ~438792 ms ago
| [        spiralCounting.dll+00004e3f = 06674e3f ] allocated     16376 bytes ~247781 ms ago
| [        spiralCounting.dll+00004ea0 = 06674ea0 ] allocated    131064 bytes ~335586 ms ago
| [        spiralCounting.dll+00004ea0 = 06674ea0 ] allocated    131064 bytes ~335582 ms ago
| [                      ???+00000000 = 00000000 ] allocated      2040 bytes ~158942 ms ago
| [        spiralCounting.dll+00004ea0 = 06674ea0 ] allocated     65528 bytes ~253896 ms ago
| [        spiralCounting.dll+00004ea0 = 06674ea0 ] allocated     65528 bytes ~167536 ms ago
| [        spiralCounting.dll+00004ea0 = 06674ea0 ] allocated     32760 bytes ~39402 ms ago
| [        spiralCounting.dll+00004ea0 = 06674ea0 ] allocated     65528 bytes ~39396 ms ago
| [        spiralCounting.dll+00004ea0 = 06674ea0 ] allocated     65528 bytes ~22290 ms ago
| [        spiralCounting.dll+00004ea0 = 06674ea0 ] allocated     65528 bytes ~22656 ms ago
| [        spiralCounting.dll+00004ea0 = 06674ea0 ] allocated     65528 bytes ~22289 ms ago
+-----> [ spiralCounting.dll / threadid = 15e0 ] total memory = 37 allocations / 1220768 bytes
```

Each DLL has each potential leaked allocation listed, followed by an overall summary, indicating total # of leaked allocations and bytes leaked. The threadid here is now displayed in hex (to be matched more easily with the debugger). In addition, the leak address for individual allocations is displayed in both relative addresses and absolute addresses.

To easily backtrack these leaked allocations to the line of code that allocated them, do the following:

(0) Open the debugger (e.g. Visual Studio).

(1) Tools -> Attach to Process... (Ctrl+Alt+P).

(2) Find cap.exe and attach to it.

(3) Pause the running program by clicking the pause button.

(4) Show disassembly if necessary. If the debugger doesn't show this by default, you can open it in Debug -> Windows -> Disassembly.

(5) Copy the address from the leaked allocation into the Address: bar. You might need to prefix the hexadecimal address with 0x (e.g. 0x06674ea0).

(6) If you compiled your module in debug mode, you should now see your C/C++ source code in line with the machine code. If you don't see line numbers, you can right click on the text and select "Show Line Numbers".

(7) You can then go to this line in your source code file. The precise line of allocation is usually the one immediately preceeding where going to that address took you.

APPENDIX F. INTERNAL VARIABLES AND ARRAYS/LISTS

1	int maxstalltime = 10000;	// max time (in ms) before a full buffer reports a stall
2	int stallrecovertime = 5000;	// max time (in ms) before a report can be repeated for the same data time
3		
4	int nosleep = 0;	// 1 = always spin and never explicitly give up timeslices
5	int worksleep = 1;	// 0 = give up timeslice but will use 100% cpu; 1 = minimal sleep time for not pure spinning
6	int useperf = 0;	// 0 = do not use performance mode, 1 = use fast performance mode, 2 = use full performance mode
7	int enablememleak = 0;	// 1 = enable memory leak detection
8	int usesched = 0;	// 1 = use user-mode scheduler
9	int debugmode = 0;	// 1 = display extremely verbose state/debug information
10	int consistencycheck = 0;	// 1 = perform state consistency checks after each operation
11	int detailedstartup = 0;	// 1 = display detailed startup information
12		
13	int PCRsize = 0;	// number of data types currently in the system
14	HANDLE PCRmutex = 0;	// thread safety mutexes for various operations
15	HANDLE PCRfinishedmutex = 0;	// mutex for notification system
16	HANDLE ctmutex = 0;	// mutex for createthread hooks
17	HANDLE heapallocmutex = 0;	// mutex for heapalloc hooks
18	HANDLE* coremutex = 0;	// mutex for individual module notifications
19		
20	int PCRtype[];	// type of this data -- this is paired with priority
21	int PCRpriority[];	// priority (subtype) for this meta
22	char* typenames[];	// type of this data in friendly, static string form
23	int PCRcount[];	// number of outputs for this same type of data (aspect/priority override)
24		
25	emPacket* PCRhead[];	// global head packet (oldest packets)
26	emPacket* PCRtail[];	// global tail packet (newest packets)
27	int PCRlen[];	// length of chain for this type
28		
29	__int64 PCRidused[];	// unique id for each packet within a type of data
30	int PCRfinished[];	// has this data type had its final packet pushed
31		
32	int* PCRnotify[];	// each metadata type points to a list of module indexes i (allmodules[i]) for quick routing/notification
33	int PCRnotifycount[];	// # of modules for each PCRnotify
34		
35	int PCRrefcount[];	// how many modules will use this input (seed for reference count)
36	int PCRoutcount[];	// how many modules will write this output
37	int PCRoutfinished[];	// how many modules will write this output that have finished
38		
39	PCRwheretype* PCRwhere[];	// ordering data types by priority for packet filter routing
40		
41	typeinfo* typeinfos[];	// friendly name bindings for user defined metadata fields
42	int numtypeinfos[];	// number of friendly name bindings within each meta type
43	char* typenames[];	// real type name (e.g., VIDEO)

```
44  int numtypes = 0;                   // number of non-priority-override types
45
46  meminfo allallocs[];                // memory leak allocation information gathering
47  int nalloc = 0;                     // number of allocations
48
49  MODULEENTRY32 ModuleList[];             // list of Windows-specific module information
50  int nmodlist = 0;                       // number of entries
```

APPENDIX G: EXAMPLE MODULE SKELETON FOR OUTPUTTING DATA TYPE "MYOUTPUTTYPE"

This module skeleton can be used as a base for other modules. To use it, a search and replace is done for 'myoutputtype' and replaced with the desired output type. By convention, lowercase and uppercase should be maintained as it appears in the skeleton. A include file should also be made, containing the data format, and then shared with other developers that will read the specified data format. Commented out portions may be uncommented depending on exact features desired.

```
1  #include "common.h"
2  #include "meta/video.h"
3  #include "meta/inside.h"
4  #include "meta/log.h"
5  #include "meta/myoutputtype.h"
6
7          // version 1.0.0, replace the description with your module's description
8      ModuleParameters ( "1.0.0", "Example module description", 0 );
9
10 typedef struct mylocals_ { // structure for data local to this module
11     emMux* mux;          // by avoiding globals, we can load separate
12     char* inputtype;     // instances of a module if need be
13     char* outputtype;
14 } mylocals;
15
16
17 emmFunction emmRegister (emModule* module, configPair* configuration, emCallbacks*
       callbacks) {
18     int i;
19     mylocals* locals;
20
21     defaultRegister();
22
23     if(!module->locals) {
24         // create a new 'locals' for thread local storage (specific to this instance of this module)
25         module->locals = (mylocals*)calloc(1, sizeof(mylocals));
26     }
27     locals = (mylocals*)module->locals;
28
29         // initialize module local variables with defaults
30
31     locals->inputtype = "VIDEO";                    // default input type
32     locals->outputtype = "MYDATATYPE";              // default output type
33     locals->mux = (emMux*)calloc(1, sizeof(emMux)); // allocate mux for this module
34
35     for(i=0;configuration[i].param;i++) {
36         if(!strcmp(configuration[i].param, "input")) locals->inputtype =
```

```
                                                                    strdup(configuration[i].value);
37          if(!strcmp(configuration[i].param, "output")) locals->outputtype =
                                                                    strdup(configuration[i].value);
38  //      if(!strcmp(configuration[i].param, "wait")) sscanf(configuration[i].value, "%f", &locals-
                                                                    >wait);       // float type
39  //      if(!strcmp(configuration[i].param, "len")) locals->len = atoi(configuration[i].value);
                                                                    // int type
40      }
41          // add input type to mux (default was "VIDEO")
42      emc->emcAddMux(module, locals->mux, locals->inputtype, 0, 0, 0, 0);
43      emc->emcAddMux(module, locals->mux, "INSIDE", 0, 0, 0, 0);     // add INSIDE type to mux
44          // register/add the mux to actual module's inputs
45      emc->emcAddInputMux(module, locals->mux);
46
47      emc->emcAddOutput(module, "LOG", 0, 0, 0);    // register/add output type
48  //  emc->emcAddOutput(module, locals->outputtype, 0, 0, 0);           // register/add output type
49
50      return 1;
51  }
52
53  emmFunction emmStart (emModule* module) {
54      return 0;
55  }
56
57  emmFunction emmData (emModule* module, emPacket* pktin, void* perf) {
58      meta_video* metavideo;
59      meta_inside* metainside;
60      emPacket* video;
61      emPacket* inside;
62      mylocals* locals;
63
64  //  meta_myoutputtype* metamyoutputtype;
65  //  emPacket* myoutputtype;
66
67      locals = (mylocals*)module->locals;
68
69          // wait for 1 packet of each data type; else, don't continue
70      if(!emc->emcMuxPacket(module, locals->mux, pktin, 1)) return 0;
71
72
73      emcGetMeta( video,   "VIDEO", locals->mux);
74      emcGetMeta(inside, "INSIDE", locals->mux);
75
76          // process data...
77
78  //  if(metainside->inside & 1) {     // if frame is inside frame...
79  // packet structure, meta structure, meta structure type, "OUTPUTTYPE", packet to inherit from
80  //
81  //          emcPreparePacket(myoutputtype, metamyoutputtype, meta_myoutputtype, locals-
                                                                    >outputtype, video);
82  // process the input data somehow, maybe include entire packet pointer or just specific fields...
```

```
83  //          processdata(metavideo->data, metavideo->width, metavideo->height, metavideo->stride);
84  //
85  //          metamyoutputtype->value1 = result1;
86  //          metamyoutputtype->value2 = result2;          // get these from somewhere, or...
87  //          metamyoutputtype->value3 = result3;          // pass emPacket* to a processdata function
88  //
89  //          emc->emcPushPacket(module, myoutputtype);   // finally, push the packet to the system
90  //
91  //    } else {
92  //          emc->emcPushDummyPacket(module, locals->outputtype, video->timestamp, video-
                                                                >final);          // push dummy packet
93  //    }
94
95              // done with this set of packets for this 1 timestamp
96        emc->emcReleaseMuxPackets(module, locals->mux, 1);
97
98        return 0;
99  }
100
101 emmFunction emmStop (emModule* module) {
102       return 0;
103 }
104
105 emmFunction emmShutdown (emModule* module) {
106       return 0;
107 }
108
109 emmFunction DllMain (HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved) {
110       defaultDllMain();
111       return 1;
112 }
```

APPENDIX H: EM-CAPTURE THRESHOLDS

The thresholds used by various EM-Capture video analysis modules for inside/outside classification are provided in the following table.

Threshold	Value	Explanation
mucosa	10	Remove dark or black pixels that are not part of the mucosa area
minimum area	1/8	At least 1/8 of the image must contain valid non-black pixels
mean-red	8	Minimum mean-red value for an inside-the-patient frame
mean-normalized-red	33	Minimum mean-normalized-red value for an inside-the-patient frame
variable-sized video analysis window	8 sec	Maximum use of temporal information within memory limits of FIFO buffer
accumulated mean-normalized-red	2400	Minimum value of mean-normalized-red of 10 sustained over 8 seconds
histdiff	5000	At least some minimum level of motion to indicate a procedure is in progress
frame classification window for procedure exit	5 min	Domain knowledge that the time between procedures is at least five minutes
outside-the-patient frame threshold for procedure exit	90%	Allow a 10% buffer for misclassification of inside/outside images
duration of black frames for fast procedure exit	90 sec	Allow for a faster procedure exit transition to occur when the scope is unplugged, in case another procedure on the same patient soon follows
variance of mean-red differences	2	Mean-red should fluctuate during a procedure
variance of mean-red differences (for exact exit)	0.05	A very strict lower bound indicates a confident procedure exit
variance of mean-normalized-red differences	0.2	Mean-normalized-red should fluctuate during a procedure
variance of mean-red differences (low threshold)	0.2	A lower, more lenient threshold for when other features values are much higher
variance of mean-normalized-red differences (low threshold)	0.05	A lower, more lenient threshold for when other features values are much higher
mean-normalized-red rise area (low threshold)	100	A lower, more lenient threshold for when other features values are much higher
mean-normalized-red rise area (high threshold)	4000	A stricter threshold for when other features are much lower
mean-normalized-red fall area	3000	Procedure exit mean-normalized-red usually has a very steep fall

Threshold	Value	Explanation
mean-red energy histogram area	64	Mean-red should fluctuate constantly and evenly
mean-normalized-red energy histogram area	16	Mean-normalized-red should fluctuate constantly and evenly
mean-red / mean-normalized-red hybrid energy histogram	32	A reasonable number of bins should contain a value of at least 5% of the highest valued bin
mean-red / mean-normalized-red hybrid energy histogram (for exact exit)	3.2	A very strict lower bound indicates a confident procedure exit
double-normalized mean-red histogram area	0.2289	At least somewhere between 20% and 25% of the 1.0 by 1.0 histogram area should be covered
analysis window for variance of feature differences	8 sec	Short-term analysis window to aid in making decisions before frames leave the FIFO buffer
histogram energy analysis windows	16 sec	A slightly larger window size than other features is more effective at combining temporal information
double-normalized mean-red histogram window	60 sec	A very large window size yields more temporal information in one feature for being extremely confident in detecting a sudden-start
mean-normalized-red difference between frames for rise/fall	-10	Allow for a small amount of noise so that a small local fall does not end a global rise and vice-versa