UNIVERSITY OF OSLO
Department of Informatics

Utilization of instrumentation data to improve distributed multimedia processing

Master's Thesis

Ståle B. Kristoffersen

Contents

List of Figures

Abstract

Instrumentation is ubiquitous in computer software today though its use in parallel processing frameworks is not widespread. In this thesis, we have developed an instrumentation framework, which we have integrated with the P2G framework. The instrumentation framework feeds a high and low level scheduler with detailed instrumentation data, while inducing a minimal of overhead. Our instrumentation framework also collects a wealth of information about the machine it runs on, including capabilities, enabling P2G to support specialized hardware. To demonstrate the feasibility of our framework, we have run a series of tests that shows promising results, both for the schedulers and developers seeking to locate a performance bottleneck, even though P2G, at the time, was not able to use this data for enhancing the decision making process

Acknowledgments

Special thanks to my supervisors, Paul Beskow, Carsten Griwodz and Pål Halvorsen, for valuable help and guidance through the entire writing of this thesis. I would also like to express my gratitude to the rest of the P2G Team for creating an awesome environment to work in.

The work on this master thesis has been done at Simula Research laboratory at IT-Fornebu. Thanks the guys and girls at the lab, for keeping the moral up, and for providing a fun place to work.

I would also like to thank my roommates; Espen Haviken, for buying food for me when I could not afford it, Jonas Karstensen for saving the planet and Roman Florinski for being the household maid.

Finally, I would like to thank my family for the endless support they have given me my entire life.

Oslo, May 16th 2011
Ståle Kristoffersen

Chapter 1

Introduction

Instrumentation is ubiquitous in computer software today; the operating system keeps statistics on each process currently executing, the download tab in your browser shows how fast a download goes and some games show you the current frame rate achieved. Such instrumentation data can be used in a number of ways. For example, schedulers use instrumentation data provided by the operating system to weight their choice when selecting the next task to run [2], i.e., how much CPU time a task demands, is used as feedback to the scheduling algorithm. By carefully selecting which tasks to run, the schedulers can attain less overhead or *fairer* scheduling, all depending on what the scheduler is trying to accomplish. Another example where instrumentation data is used is for billing purposes. Some Internet Service Providers (ISP) bill their customers based on network usage. Even though instrumentation is important, with so many possibilities, a general solution for all instrumentation has yet to surface, and may be impossible. We are therefore required to write custom solutions for most problems.

1.1 Background and motivation

Ever since the inception of programmable computers, both hardware and software have become increasingly complex. This increase in complexity is the result of the never ending quest for better performance. Already in 1965 it was noted that the number of transistors on one integrated circuit had doubled every two year from the invention of the integrated circuit in 1958, and it was predicted that this trend would continue. This prediction is what is now known as Moore's law [3]. It has been proven to be remarkably accurate, to the degree that is has become a self-fulfilling prophecy, because it is used as a target for product development [4].

To keep up with Moore's law, engineers have in the last years had to increase the number of CPU cores, because adding logic to one core got diminishing returns in terms of performance. While the increase in CPU cores in one machine does offers an increase in theoretical performance, it does not automatically provides a program written for only one CPU, more speed. So, as the hardware becomes more complex, the software has to follow suit. This leaves us with very complicated hardware and software designs, that contain numerous different parts that interact with each other in various undeterministic ways.

On the hardware front, the latest attempts to increase performance of single machines include the introduction of heterogeneous machines, where a specialized co-processor can perform some tasks at a very high speed. An example of this is seen in the current trend in decoding H.264 [5] and other video codecs by offloading as much of the processing as possible to the Graphics Processing Unit (GPU) [6–8]. The GPU can perform many of the steps required to decode a video stream at a rate much higher than the general purpose CPU the GPU is connected to. Another popular heterogeneous architecture is the Cell Broadband Engine [9], first used in the Sony PlayStation 3. It has a general-purpose Power Architecture CPU

and several RISC co-processors with 128-bit SIMD support [10].

On the software front, we are beginning to look into using more than a single machine to do the processing by distributing tasks between machines in a cluster. There are many frameworks written to utilizing a computer cluster for parallel processing, like MapReduce [11], Dryad [12] and Nornir [13]. They all need a scheduler to decide where to run each workload, but the use of instrumentation data for scheduling in a cluster of computers is still a fairly new field of study. In Hadoop [14], an open-source implementation of MapReduce, the task scheduler assumes that all tasks progress linearly. While that holds true on an homogeneous cluster, it can severely impede performance in an heterogeneous cluster. However, in [15], Zaharia et al design a new scheduling algorithm, Longest Approximate Time to End (LATE) that tries to work around the shortcoming of the default scheduler, by using instrumentation data to estimate time left for each running task, and using that as a feedback into the speculative task scheduler.

A less simple system for distributed computing, Condor [16], uses instrumentation data to decide whether or not a machine is idle. Condor does not otherwise make use of the instrumentation data for scheduling but it has another feature that is interesting; it has the ability for jobs to have certain requirements for the system it should be executed on, like operating system and hardware. This maps nicely to heterogeneous systems.

In an attempt to simplify writing and running multimedia processing on an heterogeneous computer cluster, a new framework, with a new way for expressing parallelism, is under development. It is called P2G, and as with several other frameworks, the P2G runtime consist of a central node, and several worker nodes. The central node partitions the workload, and then delegate parts to the worker nodes. Because of the way the low level scheduler on each machine can combine tasks, and the way different

implementation for a task, compiled for different architectures, can be scheduled by the central high level scheduler in P2G, we believe that it can greatly benefit from instrumentation data. We discuss P2G more in depth in chapter 3.

While the use of instrumentation data in distributed systems is not wide spread, schedulers have used instrumentation data, often called accounting data, to make decisions since the move from non-multitasking operating systems to preemptive scheduling [17]. Today, a variety of scheduling algorithms are used, but most are based on the same idea of using a multilevel feedback queue [18]. This approach has worked well on a single machine, since all of the accounting is done inside a context switch and keeps the overhead low.

1.2 Problem statement

Instrumentation is used successfully on single machine systems to not only help the scheduler, but also for many other tasks, like aiding a developer track down a performance bottleneck. We therefore want to research and develop an instrumentation framework for P2G, so that we can provide both the developers writing P2G programs and the P2G scheduler with useful and valid data. We investigate what kind of data we can provide and discuss what possible use the data might have. We look at how we should report missed deadlines, and investigate how to detect and report the different capabilities, load and status of a machine. To help P2G avoid scheduling task onto machines that are overloaded or dying, we also look at ways to detect failing machines before they fail. Our framework must not introduce much overhead, because that would render it less useful in a high performance setting.

1.3 Research method

We chose to use what in [19] is called *evolutionary prototyping*, because we are unsure about the exact requirements of the final instrumentation framework. We researched, designed and implemented a working monitoring and instrumentation framework prototype. We then integrated the prototype with the P2G framework and focused on minimizing the overhead of collecting the data and the process of making data available to both the low level and high level schedulers and to the developers.

1.4 Contributions

During the master studies, we have published a demo poster to EuroSys 2011 [20], and submitted a paper to the 2011 International Conference on Parallel Processing (ICPP-2011) [21], which is currently pending review.

We have seen that instrumentation is important for a scheduler to make the correct decisions, and the goal of this thesis has been to design and implement an efficient framework with a low overhead for data gathering. With this goal in mind, we have explored different ways to obtain timing information (Section 2.4), and created a proof-of-concept prototype (Chapter 5) proving the chosen method is viable. The instrumentation framework we have made is capable of collecting detailed system status, make precise measurements and reporting it back to to the master node, without adding too much overhead. We have also gained valuable insight into what data a scheduler could use.

1.5 Outline

Since the focus in this thesis is on providing information to a scheduler in a parallel processing framework, i.e., P2G, quite a lot of background material is needed. Both schedulers and parallel processing frameworks are quite complex, so in chapter 2, we give some background on data gathering, time sources, graphs and more. Chapter 3 introduces the P2G framework and explains how it differs from other parallel processing frameworks. It also explains the details of how P2G works.

In chapter 4, we explain the reasoning behind the design choices we have made with respect to our instrumentation daemon, and go into detail on how we set up the timers and why. Chapter 5 contains the specifics of the implemented instrumentation framework and shows how well it fits into the P2G framework.

In chapter 6 we evaluate our implementation. We go trough each data point gathered and discuss if the scheduler could make use of it in chapter 7.

Finally, we give conclusions and directions for future work in chapter 8.

Chapter 2

Background

In this chapter, we first introduce high performance clusters in section 2.1, and discuss the two methods used to split a workload to many machines. We then go through the basics of graphs in section 2.2. In section 2.3, we explain how different kinds of instrumentation work. Later in this chapter, we go into detail on how timing information can be acquired on current Linux systems, and finally, we discuss related work.

2.1 High performance clusters and types of parallelism

The usage of high performance clusters (HPC) is the only solution when a lot of processing power is needed. HPC is in use for many different purposes when applications are computationally bound, such as predicting the weather, and rendering high quality 3D graphics. In a HPC, several machines, each called a node, are connected by a network. The nodes work together to complete a workload faster than any single node could be capable of.

The first step towards parallel execution, is to decompose the workload in to self contained parts. Such workload partitioning requires a certain degree of parallelism inherent to the problem, or, at least, a possibility of expressing computationally intensive parts of the problem as a parallelizable algorithm. There are two main approaches to split a workload so that each node can process the work in parallel.

The first approach, called data decomposition, or domain decomposition, obtain its parallelism from splitting the input data and have each node work on a different part. When execution with all parts of the input data has finished, the workload is done. Exploiting this kind of parallelity is relatively easy; a program must be written that can work on a subset of the data, and then a generic framework can split the input data and distribute the parts. The framework can then start the execution on each node as fast as each node get its own data. Even with a heterogeneous computer cluster it is fairly easy for the framework to balance the work, so that no node is idling, waiting for data. Since the nodes do not share any data or state with each other, the workload completes with the same result, regardless of scheduling order and without the possibility of a deadlock.

The second approach, task decomposition, or functional decomposition, describes an approach where the data processing is split into several different computational steps, which, in turn, could be assigned to different nodes. It is potentially harder to exploit task parallelisation like this, but with a heterogeneous cluster, some tasks may be better suited for certain types of nodes, and the gain by using task parallelisation correctly can be substantial.

When a workload has been decomposed, it can be scheduled onto different nodes to leverage their combined processing power.

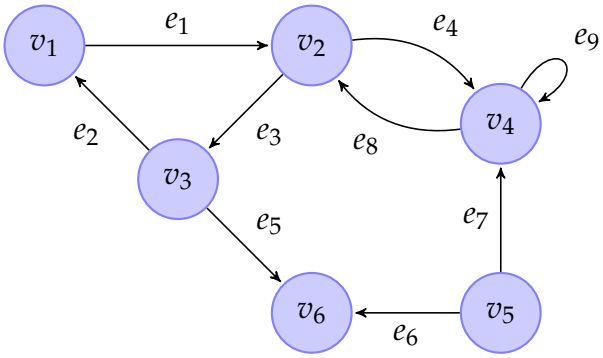
2.2 Scheduling

The job of the scheduler in a HPC is to balance workloads across the nodes in order to achieve the highest throughput. Graphs and graph partitioning is used by the scheduler to decide where to schedule different jobs, since workloads can be represented as a graph, with vertices being nodes, and edges being communication demand between nodes. How much CPU time a given task demands, or how much data it have to communicate to other nodes is often not known in advance. We want to provide the scheduler with actual measured data. The scheduler can then use that data to weight the scheduling graph. We have a small introduction into graph theory and graph partitioning in the following sections.

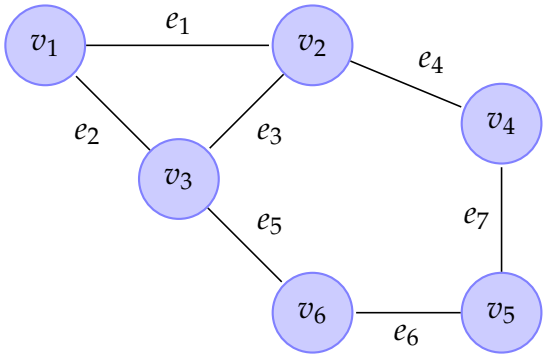
2.2.1 Introduction to graphs

A graph consists of an ordered pair $G = (V, E)$, where V is a set of vertices and E is a set of edges. The edges are simply pairs of vertices, so every edge is connected to two vertices. We have two kinds of graphs, one where the vertices consist of unordered pairs, called an undirected graph, and one where the pairs are ordered, called a directed graph. The difference is illustrated in figure 2.2.1.

In an undirected graph, there is no direction associated with a vertex. This is illustrated in figure 2.1.2. In an undirected graph, each vertex of an edge must be different, i.e., a vertex can not loop back on the same edge it originates from, so an edge like e_9 in figure 2.1.1 is not possible. In a directed graph, as shown in figure 2.1.1, you can see that the edges have a direction. An undirected graph can be changed into a directed graph by change each edge into two edges in opposing directions, like edge e_4 and e_8 in figure 2.1.1.



2.1.1: Directed graph



2.1.2: Undirected graph

Figure 2.1: Different kind of graphs

A weighted graph is a normal graph, where weight is assigned to each edge, vertex or both. To split the graph into several different partitions is called partitioning the graph, which we discuss further in the following section.

2.2.2 Graph partitioning and the need for instrumentation

Graph partitioning is used to split a graph into two or more partitions. Various criteria can be used for how the graph should be split. In our case, a high level scheduler might have a graph of a workload, weights assigned to each of the vertices, obtained through instrumentation. If different nodes communicate, the edges connecting the vertices can be weighted

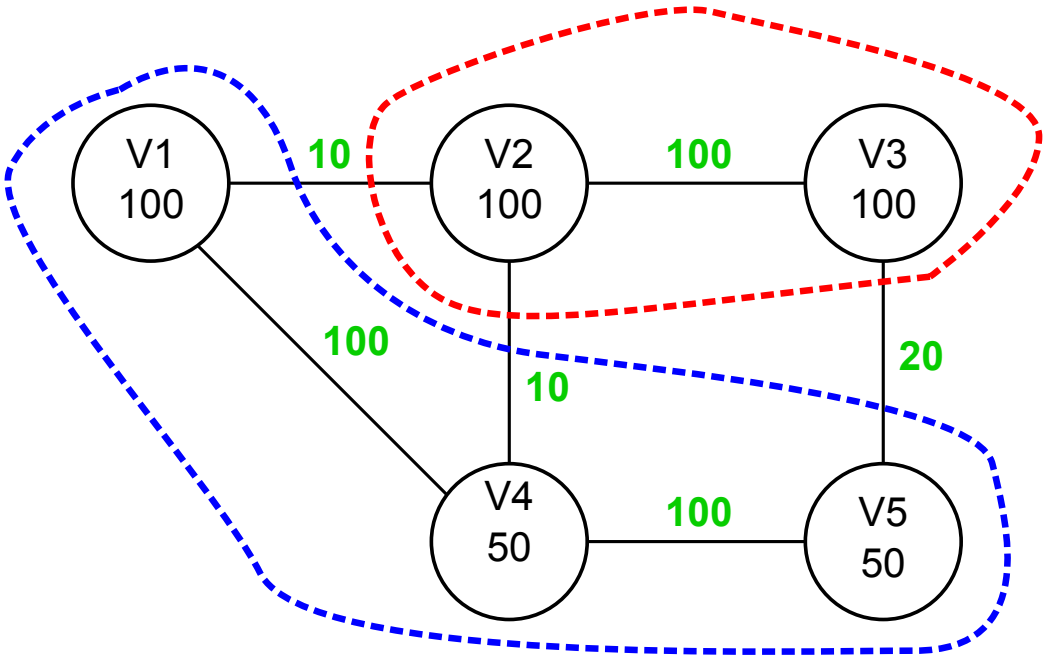


Figure 2.2: Example of a graph that is partitioned

by using instrumentation data for how much data is transmitted on each edge. From that graph, the scheduler can then partition the graph into K-partitions, where K is the number of nodes available in the HPC.

Depending on what the scheduler wants to achieve, it can partition the graph with a given criteria, for example, it can try to make each partition contain about the same weight of vertices. In figure 2.2, we can see a graph that has been partitioned into two parts, one red and one blue, both containing an equal weight of the vertics. The partition also has minimized the weight of the edges that leave or enter each partition. The same would a scheduler do to keep task that transfer a lot of data to each other on the same node in order to avoid transferring it over the network.

We do not go into how the different graph partitioning algorithms work, but interested readers may want to read more in [22].

2.3 Instrumentation

So far we have mentioned instrumentation, but not gone into any detail. We now discuss different kind of instrumentation and what they can be used for. Instrumentation can be of help for understanding the dynamic behavior of an application, both for the programmer, who wishes to improve his application, and for a scheduler that tries to find an optimal execution plan. Other uses include monitoring performance and adapt the code path taken, based on load, and admission control, that only allows new tasks to be scheduled if there is enough free capacity.

There are several different types of instrumentation; Profilers record a variety of metricses for a program, tracers record which code path are followed in a program, and timers record how long a section of code take to complete. We now explore these in detail.

2.3.1 Profiling

There are several types of program profilers; some profile memory usage, and other metricses, but we focus on those that profile by counting how many times each basic block of code is executed, called a flat profiler. An example of a profiler that can output a flat profile is GNU Gprof [23]. Profilers are a helpful tool for the programmer. It can help to determine where the optimization efforts should be focused. A common adage of software development is that 80% of the time is spent in 20% of the code [24]. Profiling can then identify where optimization is most useful. It is of little use to optimize code that is executed rarely, but of much value to speed up code that is executed often. figure 2.3, even a minor improvement in code that takes up most of the processing time is better than a big improvement in code that constitutes only a small portion of the processing time. Assume a program consisting of two separate parts,

A and B. When A is optimized so it runs 10 times faster, the whole program still takes more time to complete than if B is optimized so it runs twice as fast.

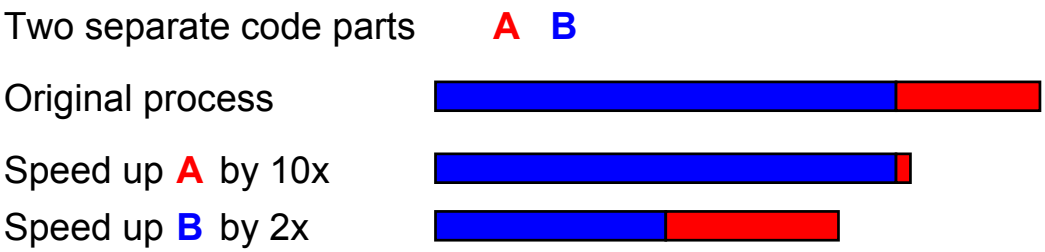


Figure 2.3: Speedup for various parts

Just in time (JIT) compiling uses profiling to estimate execution times for optimization. If a code block is executed frequently, the gain obtained by compiling that block into native, efficient code can be larger than the time lost by actually compiling it [25].

An accurate and straightforward way of doing profiling, is to insert code at the start of every code block. The code inserted at each place will have a counter assigned to it, which it increases each time it is executed. This can introduce significant overhead, because the inserted code is also executed on each branching of the original code. Alternatives have been developed, which can eliminate the recording at many of the branches, by careful analysis of the original code. If a code path that is profiled has a branch, and each arm of the branch ends up in the same position, only one of the branches needs to be recorded, since the number of times the code has gone down the other branch can be calculated by the total number of times the execution has gone into the code before the branch, minus the times it has taken the branch with the profiling code. This reduces the instrumentation overhead, while still obtaining a full profile for every block of code [25].

Take figure 2.4; It has five code blocks, connected in a simple layout. If this is the entire program, we could get away with three counters, one in

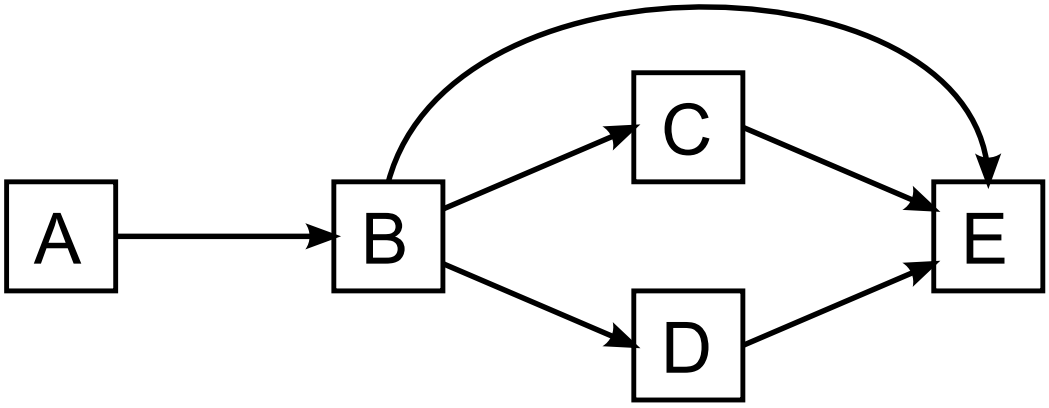


Figure 2.4: Code blocks

B, C and D, and still know how many times each block of code has been executed.

Another instrumentation approach is interrupt-based: a program's execution is interrupted at periodic intervals and the program counter is recorded. Over time, the numbers average out and should represent an accurate view of where the program is spending time. We discuss how a machine can get a periodical interrupt in section 2.4.

2.3.2 Tracing

Program tracing counts how many times each block is executed, and in what sequence they are called. GNU Plot, as discussed earlier, also support to output a call graph, showing how each code path was reached, and how many times it was reached that way. This is useful in code path analysis, which is used to audit code for possible errors. For example, the Linux kernel has a built in tracing framework, Linux Trace Toolkit next generation (LTTng) [26]. LTTng is designed to be used to debug problems that show up rarely, so it needs to be present in production code without being enabled. It works by inserting special probes in the code before

compilation. Each probe has a very low overhead in the normal case when it is not enabled. Once a situation arises that needs to be monitored, the probes can be enabled at run time and tracing information can be collected.

2.3.3 Timing

There are two ways to time a program. We can either time the execution of the whole program under one, or we can time specific code parts separately. The first is possible under Linux by invoking the *time* [27] command. The output of *time* gives us three different numbers. The first number is the time the program took to execute, it would be equal to the time a user would have to wait for it to complete. The second number is the amount of CPU time the program used, it could amount to more than the first number. The last number, is the CPU time spent in the kernel on behalf of the program.

The *time* command is only useful if we are interested in the execution time of the entire program. If the program runs indefinitely it is apparent that *time* does not work. We then have to resort to inserting timing code into the program around those pieces of code we are interested in. This would usually be a function or an inner loop to measure just how much time is spent in that particular location. The timing of one or more small, specific pieces of code is called a *microbenchmark*, and usually tells us how fast that piece of code runs.

Another way to obtain timing information during the execution of a program, is to call *getrusage* [28]. It returns various statistics on either the calling process, all of its children, or the calling thread.

2.3.4 Early failure warning

Given a computational cluster of a certain size, we are almost guaranteed that there are malfunctioning components within the cluster at any given time. An investigation performed for the Internet Archive [29] on failures of hardware in their cluster showed failure rate as high as 2% for hard disk drives (HDD), and 2.54% [30] for motherboards, CPU, memory, etc. combined. Other studies have shown failure rates for HDDs as high as 4–8% annually [31,32].

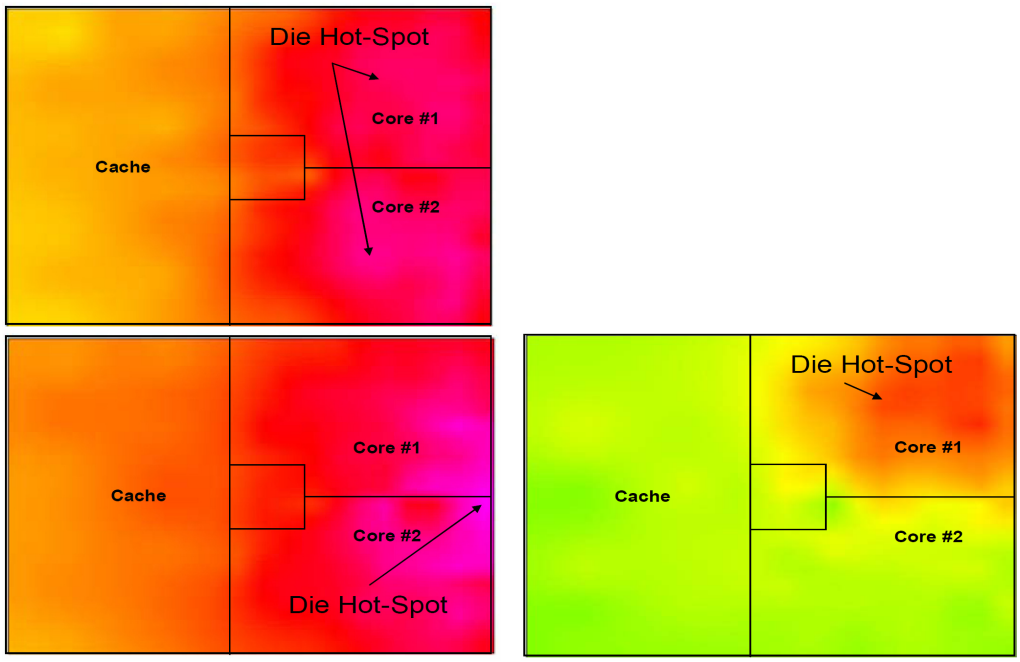
When a machine fails, its behavior can be undefined, it is therefore of interest to try to take machines out of service before they break down. There are several approaches to predict failures in different hardware [33]. We here discuss two common strategies, the S.M.A.R.T. data provided by HDDs, and the CPU temperatures provided by temperature sensors inside CPUs.

S.M.A.R.T. data

Self-Monitoring, Analysis, and Reporting Technology (SMART) is a system for monitoring the health of HDDs built into most new HDDs. It monitors several key parameters of the HDD, and tries to predict failures. Google observed over 100,000 HDDs and analyzed the data returned by those HDDs that did fail and by those that did not fail. They found some correlation between the SMART data returned and failures. A problem with SMART data is that the values returned are not standardized.

Core temperature

New CPUs contain not one, but several temperature sensors. Those additional temperature sensors are added because we want to know how



2.5.1: With different workloads 2.5.2: With a single core application

Figure 2.5: CPU Hot-spots, illustrations is from [1]

warm the hottest spot on the CPU is. For example, if the workload stresses the floating point unit (FPU), that is where the CPU is the warmest. As we can see from figure 2.5, the hot-spot of the die varies a lot depending on the workload. Each temperature sensor is checked and the warmest temperature sensor for each core is selected and that is the temperature that is returned when queried in the legacy way [1].

Many modern CPUs monitor the temperature sensors themselves, and throttle back the clock speed if a certain thermal threshold is exceeded. Some even go into a complete shutdown if the temperature increases enough.

2.4 Time sources

Modern computers have implemented several methods of acquiring timing information from the system. All current timer sources have one or more problems, they are either slow or not very accurate, and in some cases even both. Certain time sources, like the time stamp counter on many AMD processors, do not even guarantee that two successive calls return monotonically increasing timestamps [34], the last call might return a time stamp that is earlier than the time stamp returned by the second call. A programmer must be aware of such idiosyncrasies to choose a reliable and sufficiently accurate timing strategy. We now describe some of the methods, and their strengths and weaknesses.

2.4.1 Hardware timers

Hardware timers require dedicated circuitry to operate. They are usually based on a crystal oscillator, and either contain a register that can be read by the CPU or output an interrupt periodically. Different hardware timers have been implemented with various goals in mind. Hardware timers are used as a backend for all software timers. In this section we introduce the most common hardware timers available on modern computers. All of the following timers are in use today.

Intel 8253

When the IBM PC was introduced in 1981, it contained an Intel 8253 Programmable Interval Timer (PIT). While all modern IBM compatible computers contain an Intel 8253, it is no longer on a separate chip on the motherboard, since it has been integrated into the south bridge chipset. It has three channels, each implemented as a 16-bit counter that counts

down to zero. Each channel can be in one of six modes, and depending on the mode configured, it can be used for different things. Channel 0 is connected to IRQ 0, and channel 2 is connected to the PC-speaker. Channel 1 is not always present, and if present, it is not very useful because it was used to refresh the Dynamic Random Access Memory (DRAM) on early machines featuring the chip, and that functionality is not needed any more.

Access to the PIT is through four fixed I/O Ports. While relatively simple to use, it does take 3 microseconds to read it [35]. It is possible to use it as an aperiodic timer, but since it is slow to program a new timeout value, it is only used as a way to generate a periodic clock interrupt on systems without other suitable timers. The PIT is also used for tone generation on the PC-speaker.

Real Time Clock

A real time clock (RTC) was added to the IBM PC in 1984, as a way to keep track of the clock even when the system was not connected to the mains. It has a small battery to keep the clock running when the rest of the system is without power. As with the Intel 8253, the RTC is now integrated into the south bridge chipset. It can also be used to generate periodic timers, to free the PIT for aperiodic tasks. However, this is not what it was designed for and it has proven to be unreliable [36] and slow [37]. The interrupt generated by the RTC is on IRQ 8, which is of lower priority than every IRQ with a lower value. Linux only read the value at boot and after resuming from a low power state [38].

Time Stamp Counter

Every Intel x86 CPU since the Pentium has had a 64-bit register called TSC. The TSC register is incremented on every tick, and is initialized to 0 when the CPU is reset. When introduced in 1993 it was very well suited for achieving fast and accurate timing due to the fact that the tick-rate was known because it was tied to the CPU clock rate. It is low overhead since all that was needed to read it was to run one instruction, RDTSC (opcode 0F 31). When executed the result would be returned in the two 32-bit registers EDX and EAX [39]. The TSC has no way to generate an interrupt, so it can only be polled for time.

Since the introduction of multi-core CPUs and CPUs with different levels of power saving modes, some obstacles have been introduced that make it less desirable to use it as a time source.

On some multi-core CPUs, not all cores tick at the same rate. This means that over time the values diverge. This can create problems for programs that use the TSC to measure the elapsed time. If the program reads the TSC while running on one of the cores, and is then later scheduled onto another core, where it again reads the TSC, it can appear that the time has gone backwards. This is especially true for AMD CPUs [34], where AMD has released a windows driver that tries to avoid the problem by synchronizing the TSC by periodically adjusting them so they are in sync [40]. This mitigates the problem, but allows for the possibility of having the TSC value go backwards for successive reads.

When some power saving modes are activated the tick rate of the CPU might be affected, skewing the results if it was calibrated when the tick rate was different. So unless great care is taken, this is a potential source of error. On new Intel CPUs the TSC is incremented at a constant speed regardless of the current clock rate.

The Intel Pentium Pro introduced out-of-order execution, that removes the guarantee that instructions are executed in order. This means the CPU is free to reorder the instructions, which means you no longer know what you are timing. To work around this the CPU must be told to finish all previous instructions before continuing. It is accomplished by executing a serializing instruction before RDTSC. This slows down the execution, but it is still very fast [35].

Another problem encountered with the TSC is that not all x86 clones have implemented the RDTSC instruction, and even if the CPU supports it, the OS can disable it. This comes in addition to the problems described earlier, and makes the TSC unsuited as a general method of timing.

Since the rate at which the TSC increases is unknown, it can not be used as a source to calculate elapsed time, without first calibrating it by using a known timer. To calibrate it, we read the TSC at a known interval, and the tick rate can then be calculated. This only give us an approximation, that can not be more accurate than the clock used as an reference.

To summarize; The TSC, used correctly on the correct hardware, can work as expected. However, its use as a general method of timing is discouraged [41].

Local APIC Timer

The advanced programmable interrupt controller (APIC) was introduced to solve the problem for how to serve interrupts efficient on multiprocessor systems. Each CPU has its own local APIC (LAPIC). It contains a 32-bit counter and counter input register. The speed of the counter is not known, but is usually the same as the processor's front side bus (FSB) [42].

Several implementations of the APIC timers are buggy [43] [37]. It also suffers from the same problem the TSC has, in that the speed the counter

is increasing is not known. To use it, we must follow the same calibration technique as the TSC.

PM Timer

The PM timer is also known as the ACPI timer. According to the ACPI specifications [44], it is required to be present on any ACPI compatible systems. It can either have a 24 bit or a 32 bit counter, and the counter is increased at a frequency three times that of the PIT. Reading from the PM timer is also fast, only 0.7 microseconds [35]. It can be configured to raise an interrupt when the high bit changes. However, the counter value cannot be set, and when reached its maximum value it overflows and start at zero again, so it is not very flexible. The PM timer keeps running even in some of the power saving modes where some of the other timers stop or slow down, and can therefore be more reliable than other timers.

High Precision Event Timer

Intel and Microsoft developed the high precision event timer (HPET), and it was presented in Intel ICH8 chipset. It was introduced because the other available timers had flaws which made them undesirable to use. The HPET counters run at a minimum of 10 MHz, and each chip has several 32 and 64 bit counters. Each counter can have several registers associated with it, and when the value in the counter matches the value stored in one of its registers an interrupt is raised.

Access is almost as fast as the PM Timer, at 0.9 microseconds [35]. One problem with the HPET timer is that the registers for a timer only trigger when the counter is an exact match, and since there might be a large enough delay from reading the current value, to the new register value has been calculated and written back, that the counter might have passed

that number. This results in the timer not triggering before the counter has rolled over and started again, which could be a long time.

2.4.2 Software timers under Linux

We focus on software timers in Linux because that is the operating system we are using for our implementation. Other operating systems have different but similar interfaces to the timers, so much of the information here may be applied to other operating systems.

When it comes to timers, the Linux kernel has two main tasks it must accomplish. The first is to keep track of the current time and make it available through various APIs discussed later, and the other is to have a framework that allows both the kernel and user space applications to sleep for a given interval [45]. We focus on the first part, since that is what we use in our instrumentation framework later.

Software timers are the way the OS exposes the hardware timers to the applications. Some of the hardware timers can be interfaced directly from software, but even then they usually require special privileges. The TSC is a notable exception and is usually readable from user space software running as an unprivileged user.

Linux used to be based on ticks, where it scheduled a periodic timer to trigger at a given interval. Typical values have been 100, 250, 512, 1000 and 1024 times per second for different kernels. One such interrupt is called a tick. On each new tick the value of the tick-counter is increased, and since the time since the last tick was a fixed value, the kernel could increase the system time value by the same amount. When the timer interrupt occurs the current value of TSC is also saved for later use in calculation of inter-tick time. If the kernel had 100 ticks per second the system time increased at 10ms intervals.

When ticks are used and gettimeofday is called, the kernel reads the TSC again, calculated the time from the last tick and add that to the time saved as system time at the last tick. This method is error prone, because there are several race conditions and the possibilities of missed ticks made the time drift. These problems are also exacerbated when running under virtualization [42].

On newer Linux kernels, the entire timing subsystem is rewritten. The new system use something they call clocksource abstraction. In the new system, the time is calculated from scratched and returned when it is asked for and not updated during a tick. This means that kernels using the clocksource abstraction can run tick-less, i.e., they do not need to have a periodic interrupt configured.

Tick-less kernels have several advantages. They are immune to problems with lost ticks, since there are no ticks to be lost. This is a huge gain when running under virtualization, removing the need for complex logic in the hypervisor to compensate for lost ticks. Tickless kernels also eliminate a lot of unnecessary waking up, where all that is done is to update a few timers.

time

The time system call returns the number of seconds since the Epoch (00:00:00 UTC, January 1, 1970) [46]. Its resolution is in seconds and it is useless for all but the most coarse instrumentation.

gettimeofday

Gettimeofday returns a data structure containing the number of seconds since the Epoch, just like the time system call, but it also contains the

number of microseconds in the current second [47]. A problem when using the gettimeofday system call for measuring time is that it is affected by changes to the system clock. If the clock is adjusted, the time returned from a call before the adjustment and a time returned after the adjustment cannot be compared. Even with that problem, gettimeofday is widely used in a lot of software.

clock_gettime

The clock_gettime system call takes a clock-id as a parameter, that way the program can choose between multiple clocks. Recent versions of Linux have a clock-id called CLOCK_MONOTONIC_RAW which gives access to raw hardware-based time that is unaffected by changes to the system clock like NTP adjustments [48]. Because it is guaranteed to increase monotonically, linearly, and is unaffected by adjustments to the system clock, CLOCK_MONOTONIC_RAW is the preferred clock-id for timing uses. Unfortunately it can not be used to tell the current time, or use in conjunction with timeout values used like in select [49].

2.4.3 Distributed time

In a machine cluster it is desirable to have a common clock on all machines. Since that is not usually possible we are left with trying to synchronize the clocks. This is achieved by using NTP to synchronize clocks over the local network and even over the Internet. It can compensate for variable latency to the time server, and if the time server is on the local network, accuracies down to tens of microseconds can be achieved [50]. To increase accuracy, parts of the NTP clock phase-locked loop is inside the Linux kernel, running in kernel space.

Another way to distribute the clock is to use a GPS connected to each machine. Embedded in the GPS signal is a very accurate clock stream, which enables the machines to be synchronized with an maximum error of around 10 microseconds [51].

Since the local clocks on all machines drift, and in what direction and by how much is different on every machine, a daemon is usually run to continuously adjust the time to keep it in sync. Even with such a daemon running it is not advisable to rely in the distributed time being synchronized any better than to the same second.

2.5 Parallel processing frameworks

To implement instrumentation in a parallel processing framework is not a new idea. In the Nornir [13] run-time system for parallel programs, Vrba et al. implemented something they called accounting, where they could record various performance related values, like CPU time used by each process, number of context-switches, number of loop iterations while waiting to acquire a spinlock, and more. They found that this added around 0.72 microseconds in overhead, most of it from two system calls for obtaining data on per-process CPU time. However, they did not use this for any scheduling decisions, and it was mainly used to give data to the programmer about where bottlenecks are.

In Dryad [12], Isard et al. have implemented a *manager*. The *manager* can detect if some parts of the job is finishing slower than comparable parts. It can then spawn a duplicate job to make sure one slow computer does not slow the whole job down. This behavior is similar to MapReduce's *backup task*.

2.6 Summary

In this chapter we have introduced HPC and how scheduling work. We have also explained how they can use a weighted graph to schedule more efficiently. We then made the argument that we can use instrumentation data as an input to the weighting of nodes in the graph. We gave a thorough introduction to various implementation of timers, both in hardware and in software. Finally we looked at other parallel processing frameworks that use instrumentation.

In the next chapter we explain the P2G framework, and why it was created. We then explain how it can benefit from instrumentation data.

Chapter 3

P2G

In the previous chapter we introduced a lot of background information, we now show how it fits in with P2G, a framework for distributed real-time processing of multimedia data. We start by explaining the motivation to build such a framework. Then we show an example workload implemented in P2G, and explain how it would be executed. Last, we go into details about the inner workings of P2G.

3.1 Background and motivation

In recent years, it has become evident that the future development in performance of general-purpose processors will come from concurrent processing [52]. For years, the improvements in execution speed of single-threaded applications were chiefly due to ever increasing clock speeds, cache sizes and the efficiency of instruction level optimization. Around 2002, increases in clock speed stopped. You could buy a 3.06 GHz Intel Pentium 4 processor in late 2002 [53], and Intel had planned to release a 4 GHz Pentium 4 but later abandoned that plan due to *transistor leakage*

and *power density* [54]. Now, 9 years later, Intel's fastest CPU in term of clock speed has still not reached 4 GHz, instead Intel and other CPU manufactures have moved to increase the number of cores in a CPU. The trend is such that even mobile phones are equipped with multi core CPUs. We are now at a place in time when parallel processing has taken over as the main strategy for speed improvements. While multi-core CPUs offer more theoretical speed, a sequential program has to be re-written to use more than one core to exploit the possible concurrency.

Transitioning from a single to multi-threaded application is complicated and often requires domain specific knowledge of the hardware it runs on, to take full advantage of the computational capacity available. To ease this work, several frameworks have been introduced, such as Microsoft's Dryad [12] and Google's MapReduce [11].

As discussed in section 2.1, there are two main axis of expressing parallelism, and different frameworks usually only use one of the methods. For example, MapReduce uses a data parallel model, where each machine runs the same task on different parts of the data. P2G tries to improve the situation by supporting to express both data and task parallelism in a new way that is well suited for multimedia processing.

3.2 Overview

P2G is split into one master node, and an arbitrary number of execution nodes, as shown in figure 3.1. The master node contains the High Level Scheduler (HLS), Instrumentation manager and the communication manager. The HLS dispatches work to the execution nodes and the instrumentation manager gathers instrumentation data. Using the instrumentation data, the HLS optimizes where, and how, work is dispatched.

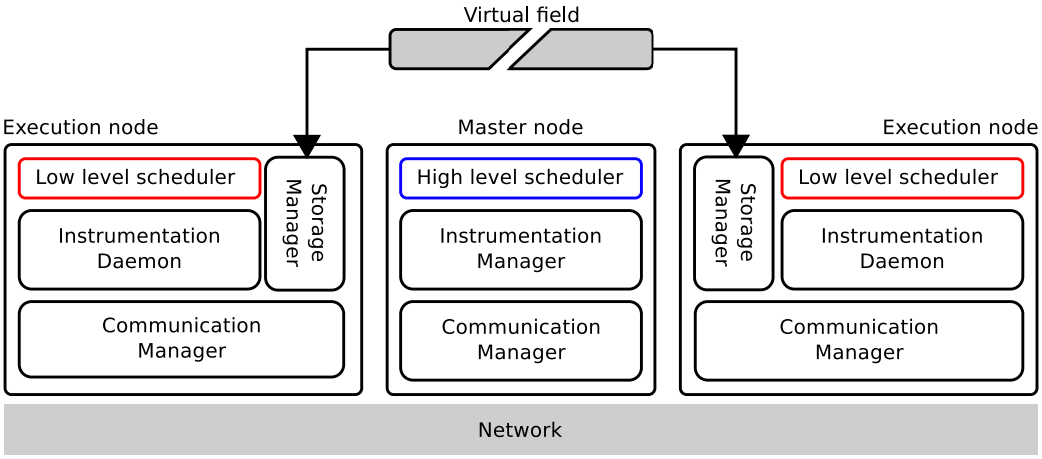


Figure 3.1: Overview of nodes in the P2G system.

Execution nodes can join the cluster at any time, and the HLS dynamically distributes the workload onto all available execution nodes. Each execution node contains a Low Level Scheduler (LLS), Instrumentation Daemon, Storage Manager and a Communication Manager. Instrumentation data that the instrumentation daemon collects is provided to the HLS via the network and to the local LLS. The LLS can, using instrumentation data to guide it, combine multiple tasks into a single execution unit, a batch, in order to reduce overhead. This is explained more in detail in section 3.2.10

A vital concept in P2G is the virtual fields, which are used to store data between each step in the processing. In reality, the virtual field is represented as a memory area on each machine that uses that virtual field, limited to the ages and indexes that the machine work on. Even when a virtual field is represented in memory, it is not guaranteed to be continuous. When more than one machine share the same virtual field, the store manager has to transport data from the machine that writes to the virtual field, to all the machines that read from the same part of the virtual field.

3.2.1 Kernel language

The kernel language developed for P2G defines how a programmer interacts with P2G, and expresses the parallelity of the program. A P2G Program consists of an arbitrary number of field declarations, and code for an arbitrary number of kernels. The field declaration defines which fields the kernels works on, and the kernel code does the work on the data in the fields.

The kernel interacts with the P2G fields through fetch and store commands. Depending on how the fetch and store commands are written, one kernel might be executed once per age, or for each index for each age.

3.2.2 Kernel code

The kernel code is the P2G code the programmer writes for their program. Each kernel consists of sequential pieces of code, that work on data stored in the virtual fields. The goal is to have each kernel express as much of an decomposition as possible, so the scheduler can combine them as it sees fit, based on instrumentation data for how long a kernel takes to execute. The code for describing a kernel is currently implemented in a C-like language that expose many of P2G's central concepts. In the next section we see an example workload written in kernel code.

3.2.3 Code example

Here is an example workload consisting of four kernels. When run the *print* kernel writes out {10, 11, 12, 13, 14}, {20, 21, 22, 23, 24} for the first age, and then {25, 27, 29, 31, 33}, {50, 54, 58, 62, 66} for the second age.

Since there is no termination condition in this workload, it continues to run and print increasing values indefinitely.

All of the following listings are usually contained in one P2G program.

field declaration

Listing 3.1: Field declaration

```
int32[] m_data age;
int32[] p_data age;
```

The field declaration defines two global fields, m_data and p_data. Depending on how the fields are declared, the fields can be write once constants, or multi aged, multi dimensional arrays. Fields are discussed further in section 3.2.5.

init

Listing 3.2: Kernel code for init

```
init:
  local int32[] values;

  %{
    for(int i = 0; i < 5; ++i)
    {
      put( values, i+10, i);
    }

  %}

  store m_data(0) = values;
```

The init kernel initializes an array with five values from 10 to 14, and stores the values in *m_data* for age 0. It has no fetch statements, and it can therefore be scheduled at start, since no fetch statements means it does not have any data dependencies that has to be met in order for it to run.

mul2

Listing 3.3: Kernel code for mul2

```
mul2:
  age a;
  index x;
  local int32 value;

  fetch value = m_data(a)[x];

  %{
    value *= 2;
  %}

  store p_data(a)[x] = value;
```

The *mul2* kernel fetches a single value from *m_data* because of index-variable x, and multiplies it by 2. It then saves the value in *p_data* in the same location, and same age. Because of its fetch statement, it has a data dependency that is not met when P2G first starts up, and it must wait for *init* to run first and fill *m_data* first, with the position given by x and a.

plus5

Listing 3.4: Kernel code for plus5

```
plus5:
```

```
  age a;
  index x;
  local int32 value;

  fetch value = p_data(a)[x];

  %{
    value += 5;
  %}

  store m_data(a+1)[x] = value;
```

The *plus5* kernel reads a single value from *p_data*, that the *mul2* kernel has put there, and adds 5 to it, it then stores that value back in *m_data* in the same location, but next age.

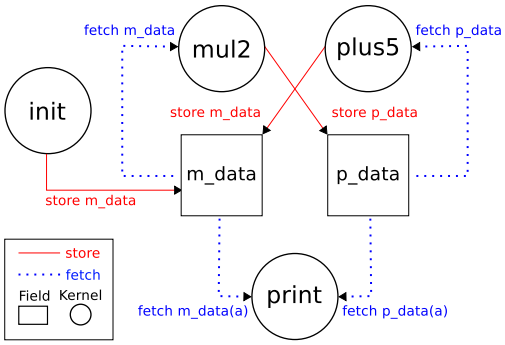
print

Listing 3.5: Kernel code for print

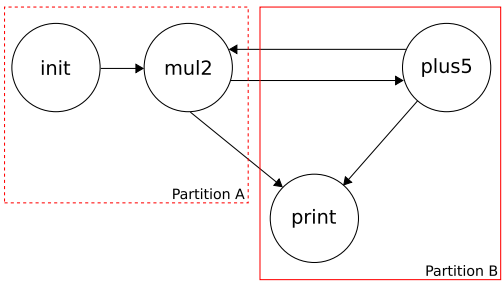
```
print:
  age a;
  local int32[] p, m;

  fetch p = p_data(a);
  fetch m = m_data(a);

  %{
    for(int i = 0;i < extent(p,0); ++i)
    {
      cout << "p: " << get(p, i);
      cout << "m: " << get(m, i);
    }
  %}
```

The *print* kernel prints out all p and m values for the current age. After the *init* kernel has run, half of its data dependencies are met,

but that is not enough. It has to wait until the *mul2* kernel has run in order for all its data dependencies to be met. The reason the print kernel fetches all the values for one age at the same time is that it lacks the *x* variable in the fetch statements.



3.2.1: Intermediate implicit static dependency graph



3.2.2: Final implicit static dependency graph

Figure 3.2: Dependency graphs

From the kernel code, P2G can build an implicit static dependency graph, see figure 3.2.1. This is because the *store* and *fetch* statements return fields and kernels gives the relationship between kernel definitions, were edges are represented by *store* and *fetch* statements on fields. Since the fields are virtual, the HLS can merge edges connecting two kernels through a field, producing the final implicit static dependency graph seen in figure 3.2.2.

3.2.4 Example walk through

When we start the execution of this program in P2G the only kernel that can run is init, since the data dependencies for the rest of the kernels are not met yet for any age. After the init-kernel starts to run, the data dependencies for some instances of *mul2* gets met and they can be scheduled to run.

For each instance of *mul2* that is run, a dependency for an instance of *plus5* is met, and that instance of *plus5* can be scheduled. Since print needs an age to be complete before it can run, all instances of both *mul2* and *plus5* for a given age must be completed before print for that same age can be run.

Plus5 needs the output from *mul2*, but writes to the input of *mul2* for the next age. *Plus5* and *mul2* therefore form a loop, which is easily identified in figure 3.2.2.

3.2.5 Field

In P2G, kernels fetch data from fields, which they perform operations on. Each field can be looked at as a global multi dimensional array, where age is one dimension, and an arbitrary number of dimensions is available for use with indexes. Fields are write-once, which means that if you have a kernel that applies a filter to all pixels in an array, it can not just write back the processed data to the same index. In the code example we had two fields, *m_data* and *p_data*. After the *init* kernel wrote to *m_data(0)*, it can not be written again for the same age. Since multimedia algorithms often write back to the same data structure it reads from P2G need to support that. The way P2G lets a kernel do that is by introducing the *Age* concept, which is illustrated in figure 3.3.

3.2.6 Age

Age is introduced as a way to get around the write-once semantic of fields. To write to the same index in a field you need to increase the age of that field. For example, when the *plus5* kernel has added 5 to the value it fetches for the first time it can not write it back to *m_data* for the same age,

because the *init* kernel has already written to it. In *plus5*, that is solved by writing to the next age which for the first invocation would be 1. Ages makes it possible to create loops between kernels like the one created by *mul2* and *plus5*, see figure 3.2.2.

3.2.7 Kernel definition

A kernel definition consists of local variable declarations, *fetch* and *store* statements, and the kernel code. The kernel code can embed native code, and that code can be as complex as necessary.

3.2.8 Kernel instance

An instance of a kernel definition is for example *mul2* with $x = 2$ and $a = 30$. Each kernel definition can lead to many kernel instances. The number of instances being executed depends on the *fetch*, and *store* statements in the kernel code. In our code example the *mul2* kernel had a fetch statement that fetched one item from one age. This makes it possible to have 5 kernel instances of *mul2* per age. However, the LLS is free to combine several kernel instances into one batch job, to limit the overhead of processing each fetch statement and the overhead of timing each kernel as seen in figure 3.2.2.

To provide feedback to the LLS about the overhead for each kernel instance we need to insert instrumentation code before and after each execution of a kernel instance.

3.2.9 Fetch and store

A fetch command can appear before the code section of a kernel. In listing 3.1, first the age and index variables are defined, and then a local 32bit variable is declared, called value. Then next, right before the code part, we have the fetch statement. It fetches only a single value from the current age a, and index x. Since the *m_data array* is of length 5, P2G, when executing, spawns up to 5 instances of the mul2 kernel per age. In section 3.2.10, we can see how P2G, using feedback from the instrumentation daemon, can decrease the number of separate executions of the kernels in order to reduce overhead.

A store command is very much like a fetch command. It has the same syntax and slice the same way as a fetch command. The only difference is that, because of the write-once semantic, store commands are always used to store to another global field, the next age, or both.

3.2.10 Dependency graph

As we already saw earlier in figure 3.2.2, P2G has a cyclic dependency graph of the workload. At run time P2G dynamically expands the dependency graph into a directed acyclic graph (DAG), as seen in figure 3.3. The move from a cyclic to acyclic graph is because of P2Gs write once semantic, where each field can have an age, so a cycle becomes a sequence of ages.

To partition the work, the HLS may use graph partitioning to distribute the load fairly onto the resources available, as seen in figure 3.2.2. With our framework we can provide information about how large the load of each kernel is, so that the HLS can use that as input to its algorithms.

With the directed acyclic dependency graph (DC-DAG) the LLS can

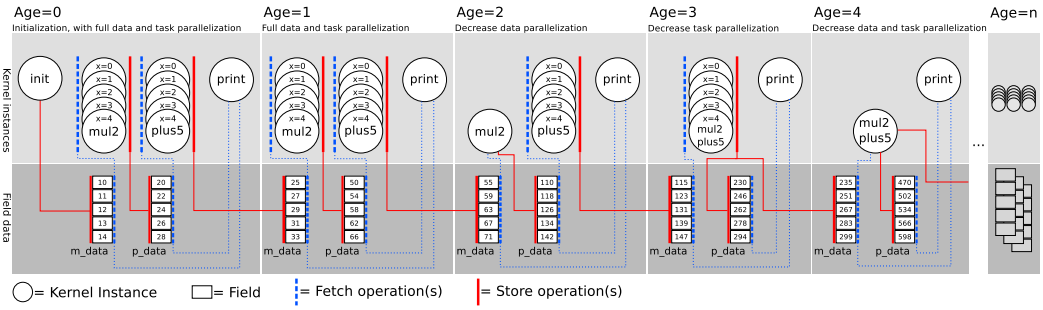


Figure 3.3: Directed acyclic dynamically created dependency graph

combine several instances of a kernel to reduce the overhead in the framework. From figure 3.3, we can see an example of how the LLS might schedule data and tasks. In age 2 the LLS has reduced data parallelity by combining the fetch statement in *mul2* to cover an entire age. The reason the LLS might do this is if the work done in the kernel is fast in contrast to the fetch statement, it then tries to reduce the overhead by fetching more data at once.

In age 3 the LLS has removed the task parallelity, and combined *mul2* and *plus5*, but kept data parallelity. This is done if *plus5* and *mul2* exchange a lot of data without doing much work, in order to reduce the overhead of network traffic. Finally, in age 4 all parallelity is removed and all data and tasks are combined.

We see that both the LLS and HLS can use data from our instrumentation framework to optimize how to execute the kernels.

3.2.11 Compiler

To transform the P2G kernel code into usable machine code the P2G framework consists of a special P2G compiler. It transforms P2G kernel code into code for different architectures. To leverage all the hard work that has been put into various compilers for various architectures the P2G

compiler transforms the kernel code into valid C or C++ code and then execute the highly optimized native compiler for each architecture, i.e., gcc for normal C-code, Nvidia's compiler for CUDA-code and any other specialized compiler for any exotic hardware.

3.2.12 Runtime

The P2G runtime consists of a daemon that you run on every machine you want to participate in the computation, and a server that controls them, see figure 3.1. The server must have all the compiled code available, so it can transmit the code to the execution nodes and tell them to run it. The runtime is responsible for dynamically load the distributed binaries and execute them safely.

The P2G runtime contains the instrumentation code, and the timing probes are inserted right before and after the execution of one kernel instance, or a batch of kernel instances.

3.3 Summary

In this chapter we have discussed the reasons behind making P2G, and shown how P2G works. We have gone into depth on how the HLS and LLS use a dynamic graph to make scheduling choices. It should be clear that in order for the LLS to be able to know when to combine single kernel instances into a batch, it needs feedback from our instrumentation framework. The HLS is also in need of feedback, so it can weight each edge and vertices in its scheduling graph, with accuracy.

In the next chapter we look at how the instrumentation framework should be design so it can provide the data P2G needs, while still being easy to

integrate and having a low overhead.

Chapter 4

Design

In this chapter, we discuss the design of our instrumentation framework for P2G, and the reasons for choosing this design. Our goal is to make a flexible framework that is modular and not coupled with P2G more than needed, while still providing valuable information to both the LLS and HLS in P2G.

We start this chapter by giving an introduction to how we want our framework to fit in with P2G. We then move on to discuss each requirement for our design. We summarize the design in section 4.7.

4.1 Introduction

The intent of this instrumentation framework is to provide instrumentation data, on a distributed platform, to a master that controls the other machines, but also to the local machine.

Our goal is to provide data to the local LLS and to the HLS. Worker nodes do not need to know anything about each other, and communication is

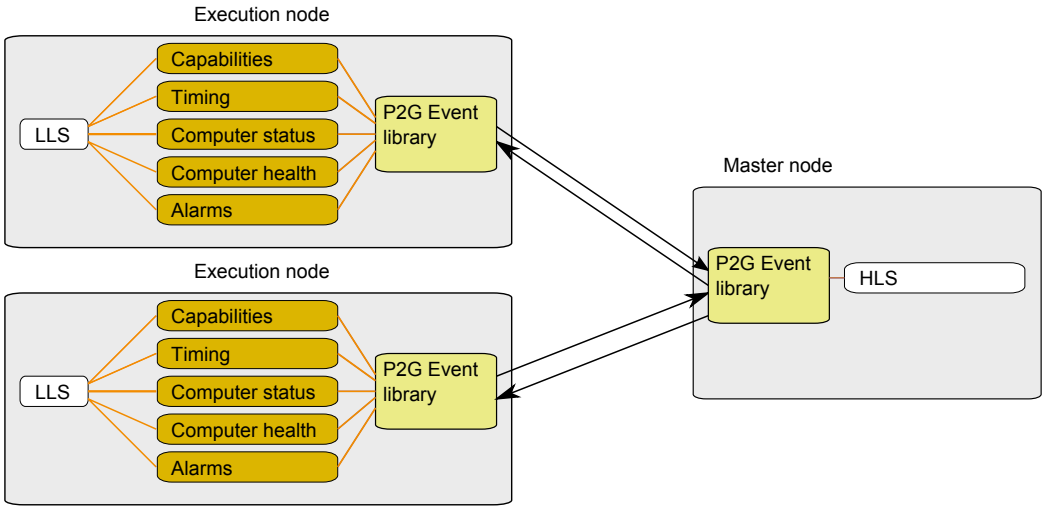


Figure 4.1: Overview of the communication between nodes

therefore only between the master node and the worker nodes, as seen in figure 4.1. Since P2G already has a communication module, we use it for all communication.

4.2 Requirements

An absolute requirement for anything that is used in a high performance computing setting is that it should be efficient, i.e., the overhead of using the instrumentation framework should be as low as possible. The main purpose is to have the scheduler make smart decisions to speed up the processing, and if the instrumentation framework consumes all the gain from applying the information it provides, there is not much point in having it. The framework should also be as non-intrusive as possible, to make the integration with P2G easier. It should also be easy to switch on/off instrumentation code, to be able to debug performance issues. Since the goal is to use the framework in the LLS to combine kernel instances into batches based on execution time, we need to measure the

execution time for each kernel instance.

We also want to use the data provided from our instrumentation framework in the HLS to combine kernels that exchange a lot of data, on one machine, so we can avoid saturating the network. For this to work, we also have to monitor the network traffic.

Another goal, is to provide data in a human readable format, to help developers locating bottlenecks in their code, so we also need a way to display the measured data.

4.3 Data gathering

We need to gather a lot of different information, and we try to keep each part separated into self contained units so they can easily be exchanged, extended or removed. Our focus is on making the solution as generic as possible, so we can adapt to different needs fast, as we do not know what kind of data our framework needs to provide in the future.

4.3.1 Capabilities

The scheduler is interested in the capabilities of the machines P2G is running on, because some of the kernels might require a CUDA capable GPU, a specific CPU-type or other specific piece of hardware. As the number of machines increases, the job to manually configure each node, becomes less viable. It is therefore vital for the scalability of P2G that the framework can detect capabilities automatically, without any human interaction. As discussed in chapter 2, certain heterogeneous systems can perform tasks at a much higher speed than a general purpose CPU. To leverage that, the program must be written and compiled for that

specific architecture. To let P2G be architecture aware and enable it to have different implementation for the same kernel, we must provide it with capabilities.

Another reason to collect capabilities is for later use to alert the HLS about conditions that should not happen. If a machine has four cores and is using only one core, it is 100% busy according to Linux. If we know that it is a quad core, we can calculate that it is still 300% idle. We might want to send an alert to the HLS informing it that this machine is not utilized well enough, while if it only has one core 100% busy is very good. We discuss how the HLS should be informed about this in section 4.4.

Linux does not have a unified way of acquiring system capabilities, so we are left to write subroutines for each piece of system info we want. Since different architectures might provide different kinds of capabilities we need to make the solution so general that more types of capabilities are easily added later.

Since, at the time of design of this module, a distributed version of P2G still did not exist, we only planned to collect basic capabilities that we felt would probably be a minimum required to make a choice for running certain workloads.

As a minimum we would need to provide information about the three basic parts that influence how suitable a machine is to run a specific workload: CPU, GPU and memory. The framework should collect information about the following:

- CPU layout
- CPU type
- GPU type
- GPU memory

- System memory

The CPU type and GPU type data enable the HLS to only assign workloads that machines can run. The CPU layout is used in our framework for load calculation. The LLS can also use the CPU layout to decide how many threads it should run. Lastly, the system memory is collected so that the HLS can assign memory intensive kernels on machines with a lot of memory.

Capabilities are not expected to change, so they can be detected once, when the capabilities are first requested. They can then be kept in memory for later use. This means that detecting capabilities is not a performance critical operation.

4.3.2 Timing and statistics

There are several aspects that we take into consideration for timing. As discussed in section 2.3, multiple approaches for instrumenting a program exists. We propose to use a simple timing and counting regime, both on the kernel definition and on the kernel code. This is because the LLS is interested in the overhead for setting up the data for a kernel and might combine several executions of a kernel definition into one batch job, as seen in figure 3.3. We time each execution of a kernel definition, and the number of times the kernel definition is executed is counted. We also do the same with the kernel code. This gives us an average execution time for each kernel and each kernel definition. By subtracting the kernel time from the kernel definition code we also get the overhead.

We modify the P2G compiler to inject our timing code at the start and end of a kernel definition and just before and after the call to kernel code in the kernel definition. This avoids the need to patch the code after it has been compiled.

Listing 4.1: Pseudo code for timing

```
_kernel_def_mul2(...)
{
  /* Start the kernel definition timer */
  timer _kernel_def_time = time.start();
  /* Set up everything for this execution of the mul2 kernel.
     */
  ...
  /* Start the kernel timer */
  timer _kernel_time = time.start();
  /* Call the kernel code */
  _mul2_kernel(...);
  /* Stop the kernel timer, and register it in the "
     _kernel_mul2"-bin */
  time.stop(_kernel_time, "_kernel_mul2");
  /* Save data and do the rest of the clean up after a kernel
     execution. */
  ....
  /* Stop the kernel definition timer and register it in the
   * "_kernel_def_mul2"-bin */
  time.stop(_kernel_def_time, "_kernel_def_mul2");
}
```

This approach gives us valuable timing data, while still providing us with some profiling data. If programmers follow the recommendation, and decompose the kernels as much as possible, we would get very fine grained instrumentation data. Even if the programmer does not follow the recommendation, P2G can only combine kernel instances, not split them up, so any finer grained instrumentation data would be useless.

We do not need to trace the program, as P2G builds a DC-DAG during runtime and that corresponds to how the program flows. It is therefore unnecessary to record the trace.

Each machine has its own timing data structure, where the collective time

spent in each of the kernel and kernel definition is saved, along with the number of times they have been run. We also provide a way for the server to signal that the client should send the current statistics back, and reset it to zero.

It is critical that the performance penalty from including the timing code is as low as possible, and it is therefore important that the code is highly optimized. Since one kernel definition can be executed concurrently on the same machine, it is also important that the code is thread safe.

4.3.3 Computer status

We also want to provide the HLS with information about the status of the machine. It is probably a bad idea to schedule more work to a machine that has run out of memory and started using swap space, and the HLS can avoid this if the status of each machine is known

This information about the computer status is periodically collected but only sent to the main server on demand. Since we can not assume P2G is the sole program running on the machine, it can take into consideration that moving a workload to a machine that already has a high load from other programs running, outside of P2G, might prove less advantageous than moving it to an idle machine.

Workloads usually consume four different resources; storage space, memory, processing power and network traffic. We therefore focus to collect statistics about those four main areas. Since memory can be swapped out, we also need to collect information about the swap utilization. The following information should be collected:

- CPU utilization
- GPU utilization

- Used swap
- Free swap
- Free memory
- Free storage space
- Network traffic

CPU utilization together with the CPU layout found in capabilities tells us if there is any free capacity left on the CPU. Used swap, free swap and free memory can tell us if the machine has used up all its memory and has begun swapping. This is normally something that is not desirable, and scheduling more work when the machine is in this state, only makes matters worse. The network traffic data can tell us if we are saturating the network link, and if it is, the HLS should use that as an input to its graph partitioning algorithm as discussed in section 2.2. The HLS can then find a way to move kernels so it can avoid having the network as a bottleneck.

4.3.4 Computer health

As discussed in section 2.3.4 there are several ways of detecting situations before they actually become a problem. We want to periodically check the local machine for possible problems, and the main server should ask all of its clients to report back if it has a condition that requires attention. The reason to not immediately raise an alarm to the main server is to avoid flooding it in the event of a misconfigured threshold value. As a proof of concept we monitor the following aspects of the computer for possible problems:

S.M.A.R.T. data

S.M.A.R.T. is used to check many parameters of the hard drives

health, including overheating, CRC errors on transfers, bad sectors and a lot more.

CPU Temperatures

We monitor the temperature of the CPU if possible. Some CPUs come with one or more integrated temperature sensors in their cores.

4.4 Alarms

When certain conditions arises we want to send an alarm to the master. The other subsystems should be able to use the alarms, to inform the master that something is wrong. When the computer status thread, discussed in section 4.3.3, detects that memory trashing has started, it should, by using the alarm subsystem, notify the HLS.

4.5 Configuration

All threshold values should have a sane default value, but the server should have the ability to reconfigure the thresholds. This is to centralize the configuration of all the nodes, and add the possibility to change the default value on all nodes in one place.

4.6 Distribution

Since we want the instrumentation framework to scale with P2G, we have opted for a pull based model, where the high level scheduler asks for instrumentation data from each node when it requires this information.

The last thing we want is to flood the high level scheduler with data, and possibly slow it down.

4.7 Summary

In this chapter we have introduced our design for the instrumentation framework, and shown how the framework would fit in with P2G. The main task for the instrumentation framework is to provide as much data as it can, while keeping the overhead low. To satisfy our goal of providing data to the HLS and LLS, the instrumentation framework must provide data about the execution time for each kernel instance, or each batch of kernel instances, in addition to statistics about the network traffic. In the next chapter we look at how this is implemented.

Chapter 5

Implementation

In this chapter we explain how we implemented the instrumentation framework designed in the previous chapter. We start this chapter by discussing what programming language we chose, and then go into the details of the implementation.

5.1 Programming Language

We have chosen the C++ [55] programming language to implement our instrumentation framework. This is because C++ provides us with the speed needed for the performance critical parts of the framework since it is a low level programming language by today's standards. In addition to providing the performance needed it still has the high level of abstraction we need to make the implementation modular and extensible.

Since P2G is also written in C++, it means that the instrumentation framework can plug straight into P2G without problems. C++ provides many abstractions that are not available in C [56], we therefore believe

that C++ is a better choice than C for splitting up each of the parts into self-contained modules.

5.2 Capabilities

We chose to save all capabilities in a standard C++ map with both the key and the value made up of strings, making it easy to dump the map in human readable format when debugging. The use of a map had the added benefit of putting the least amount of limitations on what kind of capabilities can be saved, and that fits well since we do not know what kind of capabilities we would have to implement in the future.

We implemented capabilities as its own class, since it does not rely on anything else of the framework. We made it a singleton [57] and detect all the capabilities in the constructor, so detection only happens once.

From the sysinfo [58] syscall we collect the total memory, as seen by the OS, and the total swap space available. The sysinfo syscall is very easy to use and the entire code, including converting the unsigned long value to a string took only four lines.

Listing 5.1: Using sysinfo to collect RAM and SWAP info

```
struct sysinfo curstat;
sysinfo(&curstat);
capabilities["TOTAL_RAM"] = boost::lexical_cast<std::string>(
    curstat.totalram);
capabilities["TOTAL_SWAP"] = boost::lexical_cast<std::string>(
    curstat.totalswap);
```

In listing 5.1 *capabilities* is the name of the map we fill in the capabilities in.

To detect the CPU type and layout we found it best to just parse */proc/cpuinfo*. A lot of work has been put into Linux so it can detect the

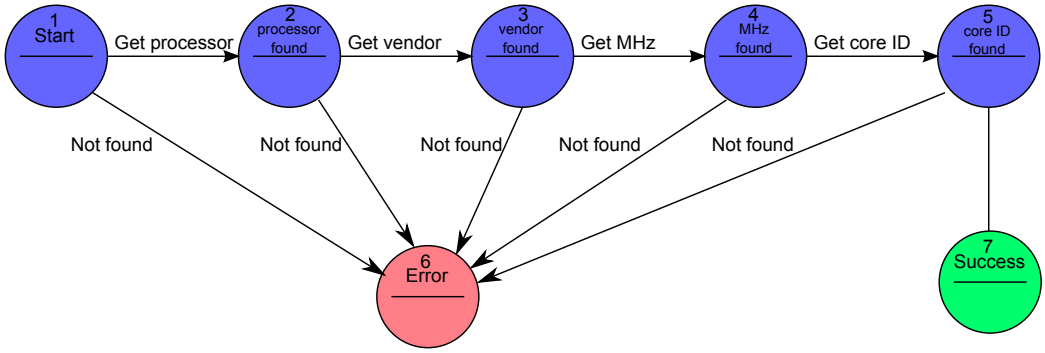


Figure 5.1: FSM for CPU mapping

CPU layout reliably, and we wanted to leverage that work by using the CPU layout that Linux exposes. To parse the CPU layout we made a simple finite state machine (FSM), see figure 5.1. After each successful core is found it is added to the capabilities map.

To show how the map over capabilities looks, we wrote code to dump it to the terminal, and the output of that is shown in listing 5.2. The CPU-lines have three trailing numbers, the first number is the physical CPU number, the second is the core number on that physical CPU and the last number is the virtual core number. In the example, both CPUs have 0 as their physical CPU number, meaning they are both in one physical package, sharing a slot/socket on the motherboard. Since the number of cores equals the number of virtual cores, the CPU does not have hyper threading enabled.

Listing 5.2: Capabilities-map for a dual core machine without any swap space

```
CPU-0-0-0-BOGO:   1994.63
CPU-0-0-0-FLAGS:  fpu vme de pse tsc msr pae mce cx8 apic sep
    mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht
    syscall nx mmxext fxsr_opt rdtscp lm 3dnowext 3dnow
    rep_good pni cx16 lahf_lm cmp_legacy svm extapic
    cr8_legacy
CPU-0-0-0-MHZ:    1000.000
CPU-0-0-0-MODEL:  AMD Athlon(tm) 64 X2 Dual Core Processor
```

```
    4200+
CPU-0-0-0-VENDOR: AuthenticAMD
CPU-0-1-1-BOGO:   1994.63
CPU-0-1-1-FLAGS:  fpu vme de pse tsc msr pae mce cx8 apic sep
    mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht
    syscall nx mmxext fxsr_opt rdtscp lm 3dnowext 3dnow
    rep_good pni cx16 lahf_lm cmp_legacy svm extapic
    cr8_legacy
CPU-0-1-1-MHZ:    1000.000
CPU-0-1-1-MODEL:  AMD Athlon(tm) 64 X2 Dual Core Processor
    4200+
CPU-0-1-1-VENDOR: AuthenticAMD
TOTAL_RAM:        4022812672
TOTAL_SWAP:       0
```

5.3 Timers

The timing framework is the most performance critical part of the entire instrumentation framework. While the other parts of the framework are executed rarely, or once each second, the timers can be used several hundred thousand times per second. We try to minimize the overhead by doing as little as possible when timing sections of code.

Looking at the code in listing 4.1 we see that the code has to get the current time twice, once at the start of the section we are timing, and once at the end of the section. In reality, the timer code is implemented as a couple of macro functions for speed reasons, see listing 5.3. Since we need a central place for all the timers to report the time, and that central place needs to be protected by a mutex to make it thread safe, we can easily run into problems of lock contention. To avoid having several threads trying to obtain the same mutex to time code the macros introduce a concept of *probes*. Each probe has its own mutex, which greatly reduce the chance of

having two threads competing for the same mutex.

Listing 5.3: High speed timers implemented as a macro

```
#define MSTARTTIME(var, fullname) static P2G::Instrumentation
    ::probe probe_##var; \
     struct timeval t_##var; \
     P2G::t->regProbe(probe_##var,(fullname)); \
     pthread_spin_lock( &(probe_##var.mutex)); \
     gettimeofday(&t_##var,NULL); \
     probe_##var.count += 1; \
     pthread_spin_unlock( &(probe_##var.mutex))
#define MSTOPTIME(var) struct timeval ts_##var; gettimeofday(&
    ts_##var,NULL); \
     time_t sec_##var; \
     long usec_##var; \
     sec_##var = ts_##var.tv_sec - t_##var.tv_sec; \
     usec_##var = ts_##var.tv_usec - t_##var.tv_usec; \
     usec_##var += sec_##var * 1000000; \
     pthread_spin_lock( &(probe_##var.mutex)); \
     probe_##var.time += usec_##var; \
     pthread_spin_unlock( &(probe_##var.mutex));
#define STARTTIME(fullname) MSTARTTIME(var, fullname)
#define STOPTIME MSTOPTIME(var)
```

The probes posed an interesting challenge, with regard to how we could avoid having lock contention. We solved this by having each probe registered at a central place, but operate on its own after it has been registered. Looking at listings 5.4 we can see that as long as p.init is true, the call just returns and no global lock is needed. This is the common path that is taken every time after the first call to regProbe for that probe, and should not induce a lot of overhead.

Listing 5.4: regProbe code

```
inline void Timer::regProbe(probe& p, const std::string& bin)
{
```

```
    if(p.init){
      return;
    } else {
      pthread_mutex_lock( &_timer_mutex);
      if (!(p.init)){
        pthread_spin_init(&(p.mutex), 0);
        proben[probeNum] = bin;
        probes[probeNum++] = &p;
        p.init = true;
      }
      pthread_mutex_unlock( &_timer_mutex);
    }
  }
```

On the first entry into regProbe, that fast path with just the return is not taken, instead the global timer mutex is acquired. Once it has been obtained, we again have to check if this probe has been initialized, because several instances of the same static probe could have raced its way into the regProbe function. If it still has not been initialized, the probes own mutex is initialized as a spin lock, and the probe is registered in the global probe array. The last thing we do before releasing the global timer mutex is to set p.init to true. If we had done this earlier, other threads might incorrectly have started using the uninitialized mutex.

When stopping the timer we can assume that the probe has already been initialized, and all that is needed is to get the current time, calculate the time difference from when the timer was started, acquire the mutex, update the total time spent in this probe and increase the count by one. We discuss the overhead of this solution in section 6.1

5.4 Computer status

As with the capabilities, we chose a map with both the key and the value made up of strings. We did this for the same reasons mentioned in section 5.2.

The computer status has been implemented in its own class, and when created it spawns a thread that has a timer that triggers each second so it can update the statistics. As with the capabilities, computer status is implemented as a singleton. This is done to avoid having more than one thread collecting status information.

We use the sysinfo system call to get data on memory and swap utilization and load, in a similar way as we got total memory and swap in section 5.2.

We found that extracting the traffic of each network card in a machine was a bit more complicated. The solution we chose was to parse */proc/net/dev* and save how many bytes were transferred. We then compared the value with the previous value we saved and then calculated the use for the last second. Special care had to be taken when the first measurement was taken, because we did not have any values to compare it with, and when the counter overflows it calculates the correct usage.

Overflowing counters is not so much a problem on 64-bit install of Linux, because it also saves the network statistics in 64-bit counters. However, on a 32-bit install of Linux, a fully utilized 10 gigabit per second Ethernet connection would wrap after 3.43 seconds. This is in part why we chose to collect info each second. Even when a overflow occurs every 3.43 second we can detect that and still record the correct usage. If the collection of statistics only ran each five second, it would be impossible to know if the counter had overflowed once or twice during that period.

CPU usage was found in a similar fashion, by parsing */proc/stat*. Among

other values, it contains the number of ticks since boot that the CPU has spent in user mode, user mode with low priority, system mode and idle. All these numbers are provided both in total and for each CPU, but we chose to only save the combined numbers. To turn those numbers into something meaningful we must save the value for later use, and compare it with the previous value so we can obtain the change for the last second. We then divide that change with the tick-rate to obtain how much of the last second was spent in each mode.

5.5 Computer health

For the SMART data, we parsed the output of the *smartctl* program. We then put the parsed data into a text map as with the other data points.

5.6 Distribution

For the distribution we, as explained earlier, decided to use the P2Gs event library. It hides the underlying socket layer from our framework, and all that is needed is a service for a given service ID. We have assigned different service IDs to the different service handlers. We have four service handlers, one that returns capabilities, one for returning timing statistics and the two discussed in more depth later.

5.6.1 Alarm service handler

The alarm service handler gets invoked when the master node connects and queries for any alarms. It then goes through each of the alarm settings

and check if any values have exceeded the threshold. A map for all exceeding values is returned.

5.6.2 Configuration service handler

The master node can send new thresholds for the alarms to the configuration service handler, which saves the new thresholds.

5.7 Summary

We have in this chapter explained the implementation of the framework outlined in chapter 4, and shown that it is feasible to provide the data we proposed. Our implementation is flexible, and since we have very compartmented code it would be able to adapt to many new forms of instrumentation without touching existing code. We now move on to the evaluation of our implementation.

Chapter 6

Evaluation

We have presented the design in chapter 4 and the implementation in chapter 5 of our instrumentation framework. It, in addition to collecting timing information for each kernel, also provides information about the capabilities of each machine and other machine metrics. Since we have discussed the rationale for both the design and implementation we do not discuss that any further here. Instead, we first quantify the execution overhead introduced by our framework, and discuss what impact that has on P2G. We then go through the data we provide, and discuss how the scheduler could benefit from that data.

6.1 Microbenchmarks

A critical part of making our instrumentation framework a success, is to have a low overhead. With a low overhead we do not increase the execution time by much. To quantify how much overhead is introduced, we have run two tests. One where we simply loop around and measure the time. This gives us the minimum overhead, since we can assume the

entire code is resident in the CPU cache and since it is run as a single thread to avoid lock contention. The other test is to insert the instrumentation code into P2G and verify that the measurements from the first test still applies.

6.1.1 Timing in a tight loop

We wrote a small program to run for an arbitrary number of iterations and in each iteration start and stop the timer as fast as possible. This gives us information about the granularity of the results, and also tell us how small the overhead is when everything is ideal. We show the source code of the program in listing 6.1.

Listing 6.1: Timer test

```
#include <instrumentation/Timer.h>
#include <boost/lexical_cast.hpp>
#include <stdlib.h>

namespace P2G{
  P2G::Instrumentation::Timer*  t = P2G::Instrumentation::
      Timer::getInstance();
  void takeTime(int iterations){
    int i;
    for(i=0;i<iterations;i++){
      STARTTIME("timer1");
      STOPTIME;
    }
    t->dumpAll();
  }

}

int main(int argc, char **argv)
{
```

```
    int iterations;
    iterations = atoi(argv[1]);
    P2G::takeTime(iterations);
    return 0;
}
```

When the program is executed, the first parameter decides for how many iterations the program should run. After the program has run through every iteration it prints out the timing statistics.

All of the following tests were run on an otherwise idle machine, with a Dual Core Intel i5 CPU, running at 2.67GHz. The machine runs Ubuntu [59], and the kernel version is 2.6.32-24-generic.

gettimeofday

In figure 6.1.1, we plotted the results of running the timer-test code for a range of iterations, all with the *gettimeofday* system call as a backend for our framework. The execution time was obtained by running the code with the *time* [46] command, and dividing that with the number of iterations. As we see in the figure, the execution time per iteration, drops fast when we increase the number of iterations. We believe this is due to the cost of starting a new process, but it is interesting to see that the measured time also goes down with more iterations. The reason for the measured time to decreased is more complicated, but could be attributed to warming up the CPU cache. Because the CPU use Intel SpeedStep technology, the CPU cores are actually running at 1.2GHz when idle. They transition to the full speed when they have work to do. The transition to full speed is not instantaneous and we believe that the dip in the measured time, after the first plateau, match up with when the CPU has time to change speed from 1.2GHz to 2.67GHz. We see that as the number of iterations increases enough, both the measured time and executed time

stabilizes. This is because the overhead of starting the executable, and the effect of both the warming of the CPU cache and the transition to full speed is negligible when averaged out over so many iterations.

A problem with using *gettimeofday* is that the granularity for measuring so small code blocks, in this example 0 lines of code, is too low. At 1 microsecond resolution, trying to measure something that takes significantly less time, is not a good idea; we get an average of less than 1 microsecond, which means most timings actually produce 0 as a result.

The total running time was 12.064 seconds for 100000000 iterations, giving an average overhead for each iteration of less than 121 nanoseconds, which at the 2.67GHz the CPU run on, equals around 322 cycles.

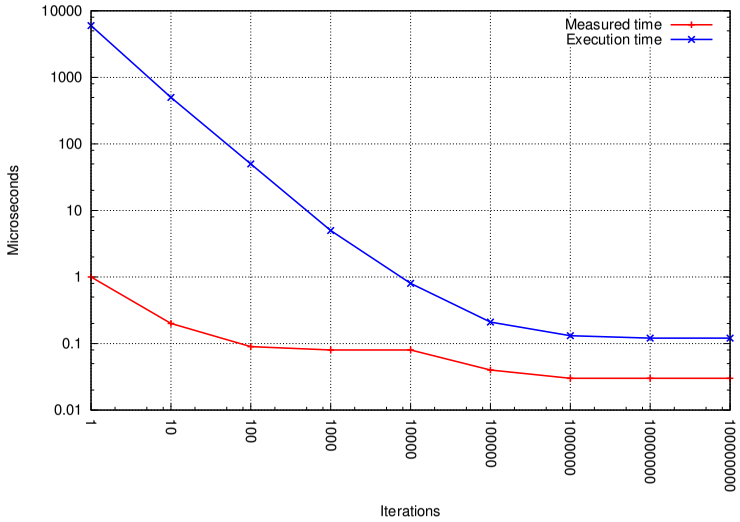
clock_gettime

When we change our framework over to using *clock_gettime* as a back-end, the speed decreases. In figure 6.1.2, we can see that it follows the same form as in figure 6.1.1, but with higher values. While *clock_gettime* is slower, it does solve the problem we had with the granularity of *gettimeofday*, since *clock_gettime* returns results with nanoseconds resolution, instead of in microseconds.

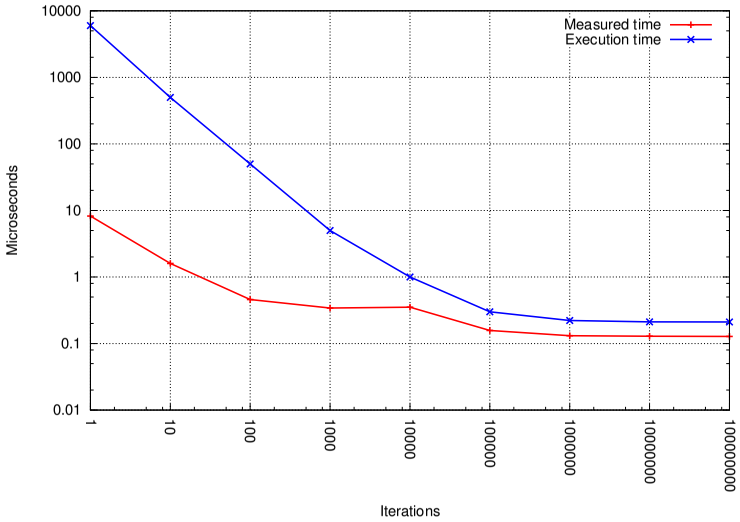
The total running time was 21.075 seconds for 100000000 iterations, which is 9 second slower, giving an average of 210 nanoseconds per iteration, or around 558 cycles. This 73% increase in execution time is clearly something we would want to avoid.

RDTSC

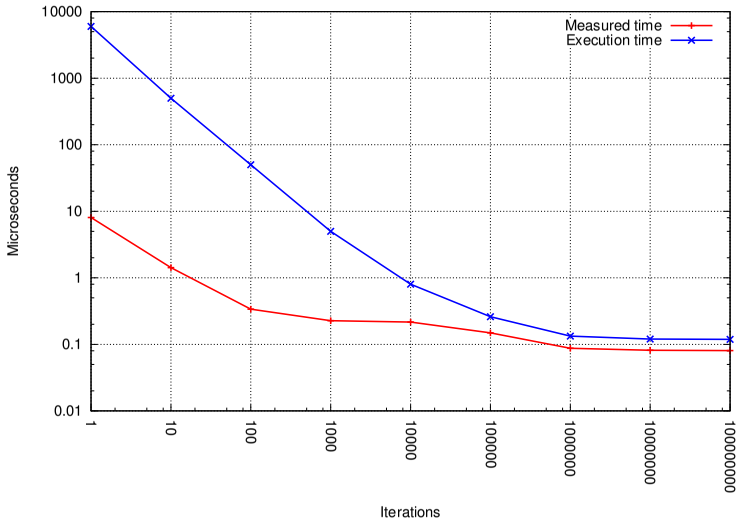
To test with the last time source, we changed over to reading the TSC. Since the TSC does not track time, we first had to calibrate how fast the



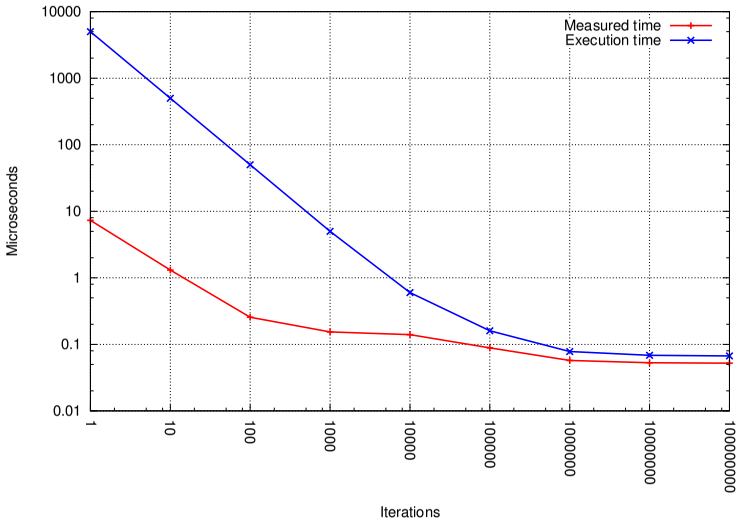
6.1.1:Timing in a tight loop, using gettimeofday



6.1.2:Timing in a tight loop, using clock_gettime



6.1.3:Timing in a tight loop, using rdtsc



6.1.4:Timing in a tight loop, using rdtsc with out serializing

Figure 6.1: Timing

TSC increases. Luckily the computer we ran our test on has the TSC synchronized across cores, and also keeps it ticking at a constant rate, regardless of how fast the CPU is clocked. Since it was a 2.67GHz CPU the tick rate for the TSC should be that, even when the CPU ran at 1.2GHz. To convert from the TSC measurements into time, we first had to calibrate our conversion factor. Since we do not have access to any of the other hardware timers when running in user space, we used the *sleep* system call, provided by the operating system, to sleep for a known amount of time. By using the following equation we get our conversion factor.

$$c = \frac{m_1 - m_2}{n} \tag{6.1}$$

c is our conversion factor, m_1 is the TSC right before the call to *sleep*, m_2 is the TSC right after the call to *sleep* and n is the number of seconds we sleep for. Since we can not assume that the operating system wakes us up after precisely the time specified in the call to *sleep*, we make n large, so the error introduced by the operating system get smaller. When we ran it with a sleep for 10 second we, as expected, got a value *very* close to the 2.67GHz the CPU is rated at.

From figure 6.1.3, we again see that it follows the same shape as the other timers. As expected the execution time was low, but interestingly not very much lower than *gettimeofday*. However, since the TSC have a very high resolution, it does not suffer from the same drawback that *gettimeofday* have with resolution, and the measurements should therefore be more accurate.

RDTSC without serializing

As discussed in section 2.4.1, a serializing instruction is needed to avoid out of order execution when reading the TSC. To see how much of an

Time source	measured	execution	diff
gettimeofday()	0.030 μs	0.121 μs	0.091 μs
clock_gettime()	0.127 μs	0.211 μs	0.084 μs
RDTSC	0.080 μs	0.119 μs	0.039 μs
RDTSC non serializing	0.052 μs	0.068 μs	0.016 μs

Table 6.1: Average for 100000000 iterations

impact that had on our execution time, we tried running it without serializing. The result, as shown in figure 6.1.4 was that it ran almost twice as fast. Unfortunately, the measurements done without serialization is unreliable, and how unreliable it is could change depending on the code executed around the RDTSC instruction.

Results

In 6.1 we have listed the numbers for 100000000 iterations, and we can see that it looks like the time not spent measuring is different on every timing method. While the overhead of our framework, and the overhead of the loop, should be constant, the system calls may spend a different amount of time before and after the timestamp was taken.

It is interesting to note how close the execution time is between *gettimeofday* and *rdtsc*. The reason for this is because *gettimeofday* actually use *rdtsc* as a clock source, only adding some glue code to calculate the difference since the last kernel tick, and converting it to microseconds. *Gettimeofday* should still be noticeably slower than using *rdtsc* by it self, since it should add the overhead of a system call. A system call normally adds a context switch, which is rather expensive, however *gettimeofday* is implemented in a special way. Since it only reads data, a shared copy of the code and the counters it needs access to is mapped into each process on the system. This reduces the overhead for that system call to the overhead of a function call,

which is much lower.

Out of the 5 different ways to acquire a time stamp we evaluated, we found that *gettimeofday* is the best choice for us, on the machine we tested on. While the resolution is not high, we can expect that each P2G kernel instance takes more time than one microsecond, and if it does not, the LLS should combine both the timing and execution of several instances into one to get lower overhead.

If *gettimeofday* was not backed by *RDTSC*, both methods would be unsuitable. The only method which is guaranteed to work correctly is *clock_gettime*, and is the one we would use for anything that is to be distributed to many machines with unknown architectures. Since we do have full control over the test setup, we use *gettimeofday* in the rest of the tests.

6.1.2 Timing of K-means clustering in P2G

The results in the previous section may not be representative for what we would get in a real scenario. To see if the numbers still applied when used to actually time something, we ran a program compiled with and without the timing code and compared the execution time.

The workload we use is an implementation of K-means clustering. It is an iterative algorithm that takes n datapoints and clusters them into k clusters. Each datapoint is assigned to the nearest cluster using euclidean distance as a metric. The new centroid for each cluster is calculated as the mean for each datapoint in that cluster, and another iteration is run. This continues until convergence has been reached. In our example however, we do not run it until convergence, because of the random input data would make comparison between runs meaningless. Instead we have

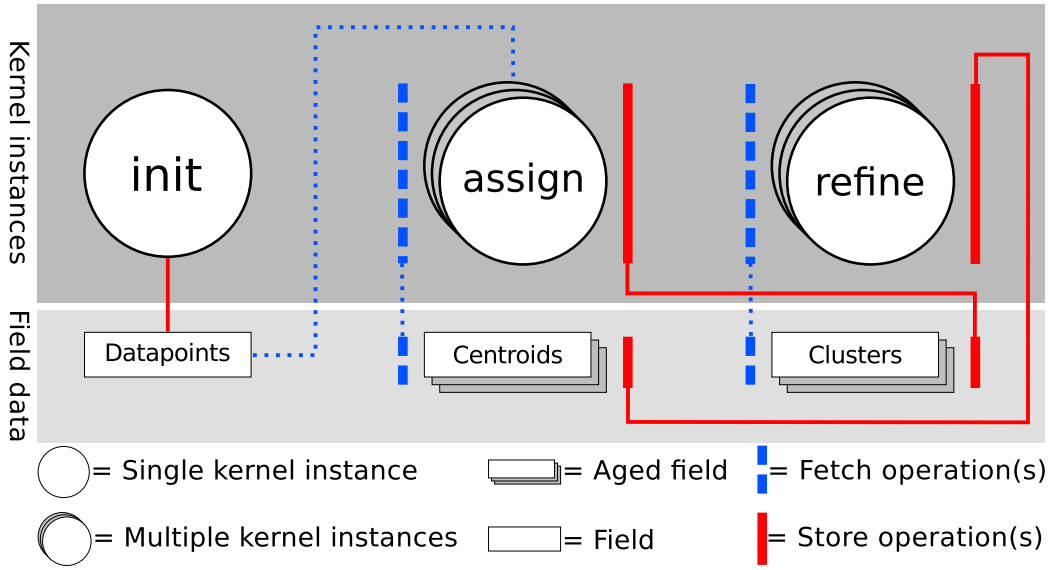


Figure 6.2: Overview of the *K*-means clustering algorithm

inserted a limit, so that the algorithm runs for a set number of iterations before terminating.

As seen in figure 6.2, the *K-means* workload consists of three kernels. First the *init* kernel generates n data points and stores them in the *datapoints* field. k of the data points are then selected at random and are written to the *centroids* field. The *assign* kernel, then reads a single point from the *datapoints* field and the last calculated *centroids* field. It then stores the data points back to the *clusters* field. Then the *refine* kernel reads a cluster and calculates the new mean, which is then stored in the *centroids* field. In figure 6.2, we can see that the *assign* and *reform* form a loop.

When we run this example workload 20 times and compare the running times with and without the timing enabled it is clear that there is some overhead introduced as expected. As we see in figure6.3, the overhead seems to increase with the number of threads above 4. We believe this increase is attributed to lock contention, which is increased when the number of threads increases beyond the number of CPU cores.

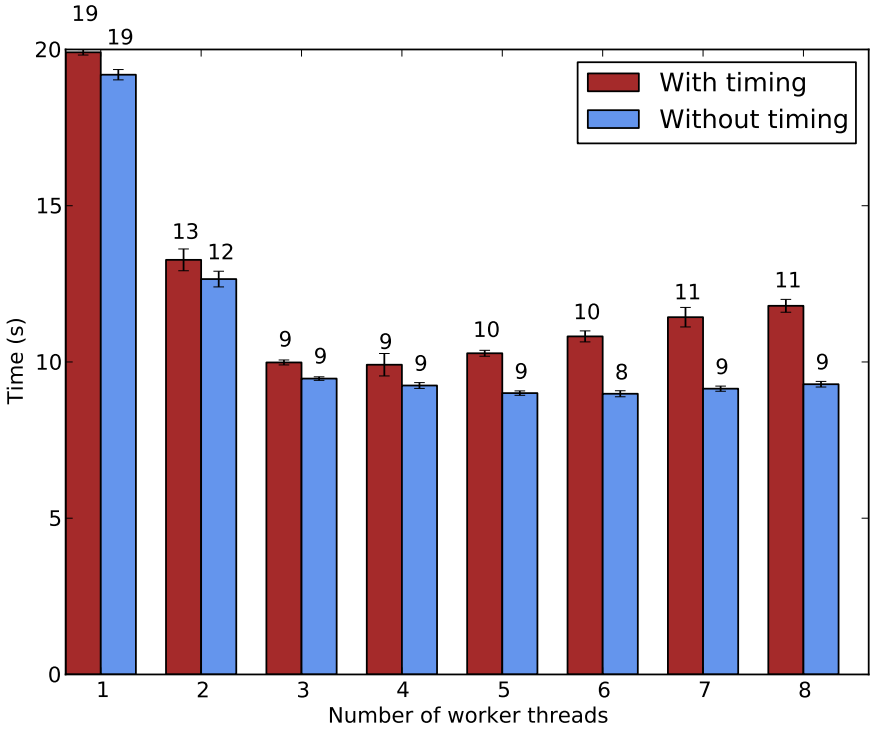


Figure 6.3: Benchmark of *K*-means clustering algorithm

From a sample run with four threads we got 4138452 measurements in 9.79 seconds, or 422722 measurements per second. As we calculated in section 6.1.1, each measurement should have an overhead of 0.121 μs. If that still holds true in our real world scenario, it would add half a second to the running time with one thread.

As we can see from table 6.2, it adds 0.72 seconds, almost 50% more than we measured in the optimal case. We believe that this is due to our

	Average	stddev	Minimum	Maximum
Without timing	19.1925	0.1695	19.0450	19.8283
With timing	19.9093	0.0897	19.7980	20.1741

Table 6.2: K-means, with and without timing running with 1 thread.

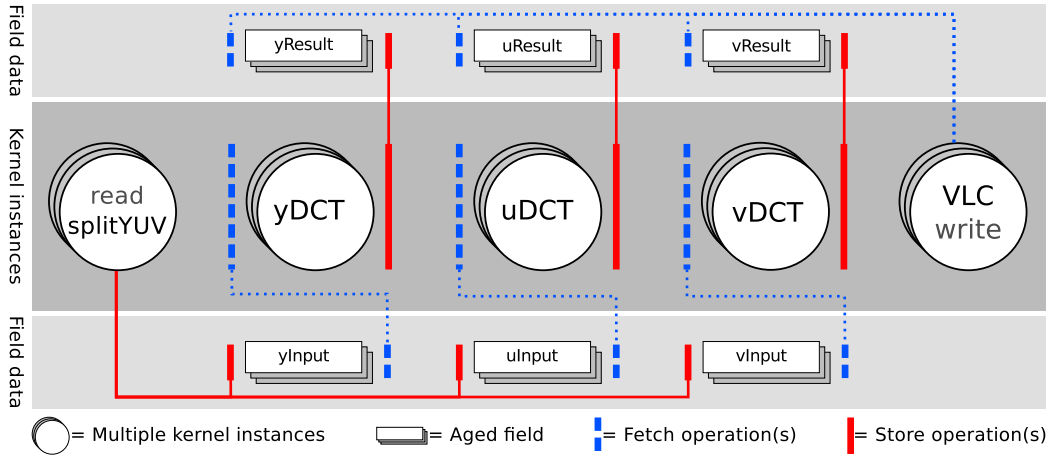


Figure 6.4: Overview of the MJPEG encoding process

instrumentation code dirtying the CPU cache and is to be expected. In all it is not at all a bad result, however, it is clear that P2Gs lack of ability to combine kernel instances and thus reducing the number of measurements is severely impacting performance.

6.2 Motion JPEG in P2G

Motion JPEG is a sequence of JPEG images compressed individually and concatenated together. It is an embarrassingly parallel workload, and how we decompositioned it is shown in figure 6.4. The *read + splitYUV* kernel, reads the raw video from disk in YUV-format. It then stores the data in three global fields, one for each DCT-kernel. Each of the DCT kernels then perform DCT on the data, creating 1584 instances of for the *yDCT* kernel, and 396 instances for the *uDCT* and *vDCT* kernels. We did not include tests for the overhead of the instrumentation data for Motion JPEG as those gave the same results as for K-means clustering.

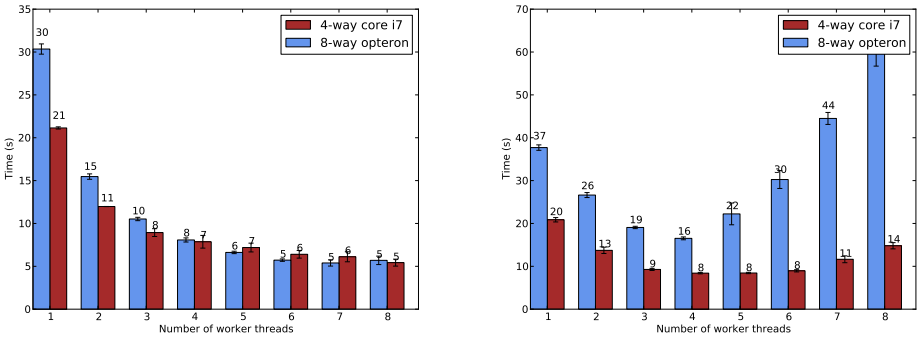
4-way Intel Core i7	
CPU-name	Intel Core i7 860 2,8 GHz
Physical cores	4
Logical threads	8
Microarchitecture	Nehalem (Intel)
8-way AMD Opteron	
CPU-name	AMD Opteron 8218 2,6 GHz
Physical cores	8
Logical threads	8
Microarchitecture	Santa Rosa (AMD)

Table 6.3: Overview of test machines

6.3 Comparison of Motion JPEG and K-means clustering

We ran the two different workloads on two different machines. The specification for each machine is listed in table 6.3.

We executed each workload on both machines and the results is plotted in figure 6.5, and the output from the instrumentation framework is listed in table 6.4 and table 6.5. These tables are examples of output provided by the instrumentation framework. This data then forms part of the information that could be utilized in descision making processes. From figure 6.5, we can see that the Motion JPEG scaled much better than *K*-means clustering when we added more threads. When we look at the data provided by the instrumentation framework, we can see that in the case of Motion JPEG, the dispatch time is not large in comparison with the kernel time. However, for *K*-means clustering, the kernel time for the *assign* kernel is very low, almost as low as the dispatch time. This makes the overhead for each kernel instance large, and many *assign* kernel instances could have been merged in order to decrease the overhead.



6.5.1:Motion JPEG 6.5.2:*K*-means

Figure 6.5: Workload execution time

Kernel	Instances	Dispatch Time	Kernel Time
init	1	69.00 μs	18.00 μs
read/splityuv	51	35.50 μs	1641.57 μs
yDCT	80784	3.07 μs	170.30 μs
uDCT	20196	3.14 μs	170.24 μs
vDCT	20196	3.15 μs	170.58 μs
VLC/write	51	3.09 μs	2160.71 μs

Table 6.4: Micro-benchmark of MJPEG encoding in P2G

Kernel	Instances	Dispatch Time	Kernel Time
init	1	58.00 μs	9829.00 μs
assign	2024251	4.07 μs	6.95 μs
refine	1000	3.21 μs	92.91 μs
print	11	1.09 μs	379.36 μs

Table 6.5: Micro-benchmark of k-means in P2G

6.4 Usefulness of data

When we started this thesis, we aimed to integrate our framework with the distributed version of P2G, and verify that the LLS and HLS could use the information we could provide. Unfortunately, the P2G development was a bit slower than anticipated, and we have been unable to test a distributed version.

We can therefore not show any improvement in P2G with the data we provide. While we do believe that both the LLS and HLS can make good use of the data, especially when the overhead of collecting it is so low. As a side note, the microbenchmarks were widely used during the development of P2G to locate bottlenecks.

6.5 Scalability and extensibility

We have seen that there are some scaling issues when moving to more threads than there are CPU cores, but we think that when the LLS gain knowledge of the system, and is informed with timing information, that those problems are solved. The framework is made with extensibility in mind, and we believe that it is very flexible. During the development we have added and removed several things with ease. Since the data is stored in a map container, and all data is handled as strings, it is up to the data consumer to parse the data provided into meaningful data. The framework therefore does not impose any artificial restrictions to what can be added.

6.6 Summary

We have measured the overhead of the timing framework, and shown that it, under ideal conditions were not large. We then moved on to real workloads, where we verified that the overhead grew, but not more than expected. We then evaluated the data provided. While the P2G schedulers still do not make use of the instrumentation data, we showed how the scheduler could use the provided data in the *K*-means clustering example, to make a decision to combine kernel instances.

Chapter 7

Discussion

In this chapter we discuss the results evaluated in the previous chapter, and various issues we have discovered with the instrumentation framework.

7.1 Hardware timers

All the hardware timers we have looked at, and used, have flaws that manifest them self as being either unreliable, slow, not universally available or any combination of those. In our test setup, the cores had synchronized TSCs, but unfortunately that is not the case for most computers today. To ensure the availability of high speed and reliable time stamps, new hardware specifications have to be introduced. The best would have been something like the TSC, but with a known, predetermined tick rate, which would be in sync for all CPU cores on a machine. In an optimal solution, the timestamp would relate directly to a given date and time, without having to calculate it based on other counters. If it was directly related to date/time it would by extension also

continue to tick at the same rate in all CPU sleep modes.

We do believe that software timers is not part of the problem, as they merely inherit the problems of the underlying hardware. If an optimal hardware timer would be exposed to the operating system, we think the current software timers under Linux would be able to leverage it.

7.2 P2G schedulers

Unfortunately, we were unable to see if our instrumentation framework could help the P2G schedulers, because the LLS is currently only implemented as simple round-robin placeholder scheduler, and the distributed version and the HLS is still not operational. This limited the options we had to validate our framework, but the results in section 6.3 shows that the data provided can be used once the schedulers get advanced enough.

7.3 Visualization

Our instrumentation framework has proven itself when it comes to visualization of how the P2G framework spend its time. As the tables in section 6.3 shows, a developer can get a pretty detailed picture of where time is spent.

Chapter 8

Conclusion

In this thesis, we have designed an instrumentation framework, based on the premise that data we provide could be used by a scheduler in P2G to optimize execution, and for developers to find bottlenecks. Here we shorty summarize the results of our work, and our most significant contributions. Finally, we discuss possible directions that future work can take.

8.1 Summary and contributions

We have implemented a working prototype for the instrumentation framework, thus validating its feasibility. Because a distributed version of P2G was still under development, and had not reached a stable state at the time this thesis was concluded, we evaluated the framework based on what kind of data it could provide to a developer and based on the overhead our instrumentation framework added with its timing probes.

During our investigation, we examined the various methods of acquiring a time stamp under Linux, and the implications of using the different timing

sources. While none of the investigated timing sources were perfect, we argued that some of them were good enough for our purpose, and that our framework, if used correctly, would provide quality data, without adding too much overhead.

The field of parallel processing is a constantly evolving area of computing. As such, we have written an article [21] about P2G, where our instrumentation framework was vital to measure and visualize where the bottleneck in P2G is. The paper is pending review, and if accepted, will appear in the proceedings of the ICPP 2011 conference, held in Taipei, Taiwan. We have also had a demo and poster [20] accepted and presented at EuroSys 2011, held in Salzburg, Austria. Again, the instrumentation framework was central for explaining the running time of different workloads in P2G.

8.2 Ongoing and future work

In addition to the functionality mentioned earlier, there are some loose ends left, which we were unable to follow. We now present some of the more promising expansions possible.

P2G is currently under heavy development, and a drawback of our work is that we have been unable to verify that the LLS and HLS can leverage data from our framework. Once the schedulers in P2G has matured enough, they should start using instrumentation data provided, and we believe it could increase throughput in P2G.

In [60], Vitter proposes an algorithm to only sample a random selection, to reduce overhead, and in [61], Cormode et al. use forward decay to reduce the importance of old measurements, without eliminating their influence. We believe these two methods can be used to provide a more complete picture of how execution times varies, without introducing too

much overhead.

Appendix

The source code and documentation is available at `http://heim.ifi.uio.no/~staalebk/m/`

Bibliography

[1] E. Rotem, J. Hermerding, A. Cohen, and H. Cain. Temperature measurement in the Intel® CoreTM duo processor. *In Proc. of Int. Workshop on Thermal investigations of ICs.*, pages 23–27, 2006.

[2] J. Aas. Understanding the Linux 2.6. 8.1 Scheduler. *Silicon Graphics, Inc.(SGI), Tech. Rep*, 2005.

[3] G.E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38:114–117, 1965.

[4] C. Disco and B. Van der Meulen. *Getting new technologies together: Studies in making sociotechnical order*. Walter De Gruyter Inc, 1998.

[5] Thomas Wiegand, Gary J. Sullivan, Gisle Bjntegaard, and Ajay Luthra. Overview of the H.264/AVC video coding standard. *IEEE Trans. Circuits Syst. Video Techn*, 13(7):560–576, 2003.

[6] W. De Neve, D. Van Rijsselbergen, C. Hollemeersch, J. De Cock, S. Notebaert, and R. Van de Walle. GPU-assisted decoding of video samples represented in the YCoCg-R color space. In *Proceedings of the 13th annual ACM international conference on Multimedia*, pages 447–450. ACM, 2005.

[7] S. Ryoo, C.I. Rodrigues, S.S. Baghsorkhi, S.S. Stone, D.B. Kirk, and W.W. Hwu. Optimization principles and application performance

evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82. ACM, 2008.

[8] B. Pieters, D. Van Rijsselbergen, N. De, et al. Performance Evaluation of H. 264/AVC Decoding and Visualization using the GPU. In *Proceedings of SPIE, the International Society for Optical Engineering*. Society of Photo-Optical Instrumentation Engineers, 2007.

[9] H.K. Stensland, H. Espeland, C. Griwodz, and P. Halvorsen. Tips, tricks and troubles: optimizing for cell and gpu. In *Proceedings of the 20th international workshop on Network and operating systems support for digital audio and video*, pages 75–80. ACM, 2010.

[10] H.P. Hofstee. Power efficient processor design and the cell processor. In *Proc. of the 11th International Symposium on High-Performance Computer Architecture*, pages 258–262, 2005.

[11] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proc. of USENIX OSDI*, 2004.

[12] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proc. of ACM EuroSys*, pages 59–72, New York, NY, USA, 2007. ACM.

[13] Z. Vrba, P. Halvorsen, C. Griwodz, P. Beskow, and D. Johansen. The Nornir run-time system for parallel programs using Kahn process networks. In *2009 Sixth IFIP International Conference on Network and Parallel Computing*, pages 1–8. IEEE, 2009.

[14] T. White. *Hadoop: The Definitive Guide*. Yahoo Press, 2010.

[15] M. Zaharia, A. Konwinski, A.D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments.

In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 29–42. USENIX Association, 2008.

[16] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience*, 17(2-4):323–356, 2005.

[17] K. Thompson. UNIX implementation. *The Bell System Technical Journal*, 57(6):1931–1946, 1978.

[18] L. Kleinrock and RR Muntz. Processor sharing queueing models of mixed scheduling disciplines for time shared system. *Journal of the ACM (JACM)*, 19(3):464–482, 1972.

[19] A.M. Davis, E.H. Bersoff, and E.R. Comer. A strategy for comparing alternative software development life cycle models. *IEEE Transactions on Software Engineering*, pages 1453–1461, 1988.

[20] P.B. Beskow, H. Espeland, H.K. Stensland, P.N. Olsen, S. Kristoffersen, E.A. Kristiansen, C. Griwodz, and P. Halvorsen. Distributed real-time processing of multimedia data with the p2g framework. EuroSys, 2011.

[21] H. Espeland, P.B. Beskow, H.K. Stensland, P.N. Olsen, S Kristoffersen, C. Griwodz, and P. Halvorsen. P2g: A framework for distributed real-time processing of multimedia data. *Submitted to: International Conference on Parallel Processing*, 2011.

[22] R. Diestel. *Graph theory*. Graduate Texts in Mathematics 173, 4th edition, 2010.

[23] J. Fenlason and B. Baccala. Gnu-gprof: user manual. *Free Software Foundation, November*, 1998.

[24] D. Port, R. Kazman, H. Nakao, and M. Katahira. Practicing What is Preached: 80-20 Rules for Strategic IV&V Assessment. In *Proceedings*

of the 2007 IEEE International Conference on Exploring Quantifiable IT Yields, pages 45–54. IEEE Computer Society, 2007.

[25] T. Ball and J.R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4):1319–1360, 1994.

[26] M. Desnoyers and M. Dagenais. LTTng: Tracing across execution layers, from the Hypervisor to user-space. In *Linux Symposium*, 2008.

[27] Linux. *man 1 time*.

[28] Linux. *man 2 getrusage*.

[29] Internet archive is a non-profit digital library offering free universal access to books, movies & music, as well as 150 billion archived web pages. `http://www.archive.org/`.

[30] T. Schwarz, M. Baker, S. Bassi, B. Baumgart, W. Flagg, C. van Ingen, K. Joste, M. Manasse, and M. Shah. Disk failure investigations at the internet archive. In *Work-in-Progess session, NASA/IEEE Conference on Mass Storage Systems and Technologies (MSST2006)*. Citeseer, 2006.

[31] G.F. Hughes, J.F. Murray, K. Kreutz-Delgado, and C. Elkan. Improved disk-drive failure warnings. *Reliability, IEEE Transactions on*, 51(3):350–357, 2002.

[32] E. Pinheiro, W.D. Weber, and L.A. Barroso. Failure trends in a large disk drive population. In *Proceedings of the 5th USENIX conference on File and Storage Technologies*, pages 17–28, 2007.

[33] F. Salfner, M. Schieschke, and M. Malek. Predicting failures of computer systems: A case study for a telecommunication system. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, page 415. IEEE, 2006.

[34] Advanced Micro Devices Inc. Amd tsc drift solutions in red hat enterprise linux®. http://developer.amd.com/documentation/articles/Pages/1214200692.aspx.

[35] J. Wassenberg. Timing Pitfalls and Solutions. http://algo2.iti.kit.edu/wassenberg/timing/timing_pitfalls.pdf, 2007.

[36] OSDev. Rtc. http://wiki.osdev.org/RTC.

[37] Microsoft. Guidelines for providing multimedia timer support. http://msdn.microsoft.com/en-us/windows/hardware/gg463347.aspx.

[38] Linux. *man 4 rtc*.

[39] Intel Corporation. Using the RDTSC Instruction for Performance Monitoring. Technical report, tech. rep., Intel Corporation, 1997.

[40] Advanced Micro Devices Inc. Amd dual-core optimizer version 1.1.4. http://support.amd.com/us/Pages/dynamicDetails.aspx?ListID=c5cd2c08-1432-4756-aafa-4d9dc646342f&ItemID=153.

[41] Chuck Walbourn. Game timing and multicore processors. http://msdn.microsoft.com/en-us/library/ee417693%28VS.85%29.aspx.

[42] VMWare. Timekeeping in vmware virtual machines. http://www.vmware.com/files/pdf/Timekeeping-In-VirtualMachines.pdf.

[43] R. Gioiosa, F. Petrini, K. Davis, and F. Lebaillif-Delamare. Analysis of system overhead on parallel computers. In *Signal Processing and Information Technology, 2004. Proceedings of the Fourth IEEE International Symposium on*, pages 387–390. IEEE, 2004.

[44] Hewlett-Packard Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd., Toshiba Corporation. *Advanced Configuration and power interface specification, 4.0a*.

[45] Daniel P. Bovet and Marco Cesati. *Understanding the linux kernel*. O'Reilly, 3rd edition, 2005.

[46] Linux. *man 2 time*.

[47] Linux. *man 2 gettimeofday*.

[48] Linux. *man 3 clock_gettime*.

[49] Linux. *man 2 select*.

[50] D. Mills, J. Martin, J. Burbank, and W. Kasch. Network Time Protocol Version 4: Protocol and Algorithms Specification. RFC 5905 (Proposed Standard), June 2010.

[51] A. Pásztor and D. Veitch. PC based precision timing without GPS. *ACM SIGMETRICS Performance Evaluation Review*, 30(1):1–10, 2002.

[52] H. Sutter. The free lunch is over. *Dr. Dobb.s Journal*, 30(3), 2005.

[53] Intel Corporation. Microprocessor quick reference guide. `http://www.intel.com/pressroom/kits/quickrefyr.htm`.

[54] E. Bangeman. Intel pulls the plug on 4GHz Pentium 4. *Ars Technica*, 2004.

[55] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000.

[56] Brian W. Kernighan. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[57] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*, volume 206. Addison-wesley Reading, MA, 1995.

[58] Linux. *man 2 sysinfo*.

[59] Canonical Ltd. Ubuntu. `http://www.ubuntu.com/`.

[60] J.S. Vitter. An efficient algorithm for sequential random sampling. *ACM transactions on mathematical software (TOMS)*, 13(1):58–67, 1987.

[61] G. Cormode, V. Shkapenyuk, D. Srivastava, and B. Xu. Forward decay: A practical time decay model for streaming systems. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 138–149. IEEE, 2009.