Towards multi-core execution time distributions

Michael de Lang
michael.delang@student.uva.nl

August 18, 2019, 28 pages

Supervisor: Sebastian Altmeyer
Host organisation: Universiteit van Amsterdam

Contents

Abstract

Embedded software is increasingly run on multi-core hardware, resulting in a more difficult to estimate worst case execution time. Multi-core programs, unlike single-core programs, have extra factors that influence execution times among which memory and disk contention and shared caches. Our research focuses on three things in this problem space: What trade-offs exist in simulating only part of the state-space, which simulation mode of *gem5* can best be used in large state-space settings and what is the feasibility of using *gem5* in multi-core execution time analysis. Building on top of Edify, we iterate over variables influencing execution times and the order in which tasks are executed and which cores. We cut these into two separate steps: the variables influencing times and then permuting over task order, using the output of the first as input for the second step. This reduces the state-space to an acceptable level and allows us to then validate the accuracy by executing another variables influencing times step with the output of step 2 as input for this validation. The software we use in our simulations is *PapaBench*, a free Real-Time benchmark, designed specifically for dual-core benchmarking. For simulating, we use *gem5* and distribute the runs over multiple cores on a supercomputer. In System-call Emulation mode, *gem5* is easily distributed. To determine the fitness of System-call Emulation mode for our research, we simulate *TACLeBench* in System-call Emulation mode and Full System mode. We find that System-call Emulation mode offers the most deterministic results, has very explainable differences with Full System mode and is in most cases nearly as accurate as Full System mode. Moreover, our results for the state-space analysis show that the capability to make a lot of simulations is available in modern day technology, however it is not enough to do full state-space simulations. Furthermore, reducing the state-space as proposed leads to less accurate results. Out of these experiments we also find some shortcomings in *gem5* and our approach: unable to create custom interrupts and hooking simulated hardware to trigger these, lack of a shared memory mechanism in *PapaBench* and allowing task order permutations that would not occur in normal usage of *PapaBench*.

Chapter 1

Introduction

Worst case execution time measurements are made because some systems, such as automotive or aerospace systems, require tasks to be completed within a certain time frame. There are two general classes of methods to estimate WCET. One is to measure actual execution times on either real hardware or in simulations, this is called dynamic timing analysis. However, it is difficult to determine if the measured execution times are worst-case or not, as well as the hardware required to do enough measurements in an acceptable time-frame is high. The other is analysing WCET based on static analysis, for example Control Flow Analysis. Static analysis often over-approximate the WCET, resulting in a decision to use more powerful than needed hardware to ensure execution time demands are being met.

Moreover, most WCET analyses happen on single-core systems, while multi-core systems are becoming a must even in embedded systems. The state-space for single-core systems with WCET depends on scheduling policy, system input, system state and system interference. Multi-core WCET estimation adds more to this list: which tasks are scheduled on which cores, sharing (some) caches and more interference between the cores due to memory and I/O contention. Given these variables, it is easy to conclude that the state-space is simply too big to use a brute-force approach into getting a WCET distribution.

In this thesis, we re-examine this conclusion. Due to the expanded availability of machines with high CPU core counts in the cloud and having simulation tools available to simulate multi-core systems, we believe that there is a possibility that the conclusion does not hold anymore. Additionally, we investigate a method of reducing the state-space using the *gem5* simulator [1] and the *PapaBench* benchmark software [2].

1.1 Problem Statement

The problem we study is the feasibility of measuring WCET on multi-core systems, without knowing which inputs leads to the absolute worst case possible. Previous work, called *Edify*, done on generating a WCET distribution using *gem5* by varying the input of programs [3] limited itself to single-core systems and used the single-core portions of benchmark suites *TACLeBench* [4] and *EEMBC Autobench* [5]. *Edify* works by taking the source code of a task, running static analysis to extract the relevant input variables impacting WCET and compiling all possible programs with these inputs and running them all. This method is close to the embarrassingly parallel category, which combined with the high cpu core count virtual machines available today, suggests that using this brute-force method is feasible for larger state space problems such as measuring WCET on multi-core systems. We therefore focus on the following research questions:

- RQ1: How to tackle the state-space problem?

One of the bigger technical problems in determining an accurate execution distribution for software is the state-space. We would like to find an approach that either limits the state-space or increases the amount of state-space that can be processed per time frame. We approach this by both creating a framework that allows us to run a lot of simulations in parallel as well as investigating the impact of reducing state-space by isolating execution time influencing factors into separate runs and using the output of one run as the input for the next run.

- RQ2: Which emulation mode of *gem5* can best be used in large state-space settings?

gem5, explained more in detail in Section 2.2, contains two modes of simulation. System call emulation mode (SE) and Full System (FS) emulation mode. Both of these have advantages and disadvantages. SE mode routes system calls to the host system, trading in accuracy of the simulations for speed of simulations by not needing to boot an entire kernel first inside the simulation. FS mode does boot the entire kernel and offers a full Linux environment to do simulations with. In terms of dealing with the state-space problem, which of the two modes, both or neither, are suitable?

- RQ3: Does *gem5* currently provide the facilities to do a full WCET analysis for *PapaBench*?

PapaBench, as noted in Section 2.2, provides a good base to figure out if it is possible to fully emulate everything that involves running embedded software. Which facilities are required by *PapaBench* to fully simulate it and does *gem5* offer them?

1.2 Solution Outline

Since not all inputs that have an effect on WCET distribution are known or easily isolated, we provide a best-effort baseline to compare against. To create simulations, we use the clock-cycle accurate *gem5* simulator, which has some research behind it regarding accuracy in various settings. Using single- and multi-core benchmarks, we then evaluate which settings of *gem5* create the best match between parallel computing, accuracy and ease of use. Additionally, we implement a framework to dissect *PapaBench* and use it to run simulations on a supercomputer to see how feasible creating an execution time distribution is for multi-core programs. This framework focuses on two big state-space exploding factors: determining the branches in code and determining the order of tasks that lead to the highest execution times. To do this, we modify *PapaBench* to produce an executable with an API which determines at run-time what branches of tasks and in which order tasks should be executed. Our framework then has a two-step approach: first simulate all tasks individually with permuting over all branches. This results in an output with the values of variables which lead to the highest amount of instructions measured in *gem5*, per separate task. The second step is to use these values in the next run, where the framework permutes over the order in which tasks are executed. To assess the accuracy of dividing the state-space into two steps, we then do a third step with our framework, to use the order of tasks with the highest execution time found in the previous step and permute over the possible branches again. If there are no simulations with a higher execution time in step 3, than what was found in step 2, accuracy is deemed high.

Chapter 2

Background

2.1 WCET

As WCET has potentially multiple meanings, talking about WCET can lead to confusion. Wilhelm et al. [6] have succinctly grouped the various meanings as can be seen in Figure 2.1. When measuring WCET using dynamic timing analysis, one always has to keep in mind that these methods are not 100% accurate. Therefore, a margin of error has to be added to every measurement or calculation. That what is measured is then increased with this margin to attain an upper timing bound, of which is then assumed that it is sufficiently high to be more than the actual possible WCET. Static timing analysis is assumed to be pessimistic by construction, wherefore this error margin is not applied.

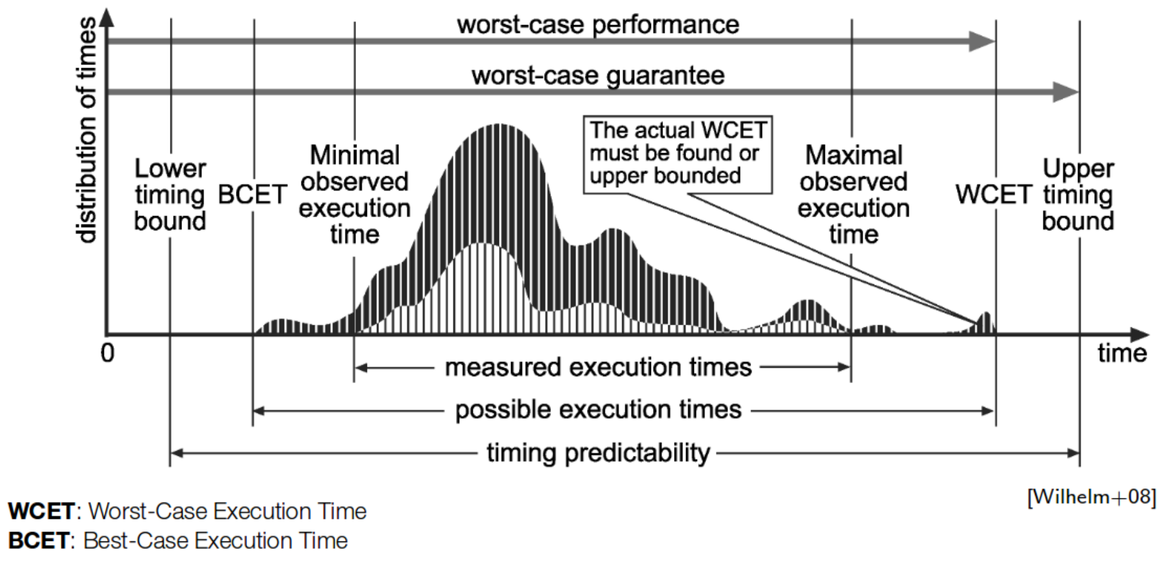


Figure 2.1: Illustration of what WCET and BCET mean, taken from [6]. The white curve represents a subset of measured executions, the darker curve represents the times of all executions.

In general, there are two categories of trying to determine the WCET of a system. One category, *dynamic timing analysis*, is the measuring of execution times through simulating a program or executing it on hardware. To ease the computation requirements of measurement, often only a subset of the actual possible executions are taken into account. Newer measurement-based approaches measure different parts of a task and combine them to give better estimates, though these methods rarely guarantee upper bounds on execution times. As can be seen in Figure 2.1 in the white curve, this approach leads to overestimated BCET times and underestimated WCET times. The other category is an approach not of measuring but of combining the source code of a task or system with some (abstract) model of the system and aims to obtain upper bounds from this combination. An example of a method in this category is Model Checking. This category is identified as *static timing analysis*.

2.2 Applications used by our framework

PapaBench is a benchmark that aims to provide a close to real usage benchmark scenario. It does so by simulating a drone program with two CPUs and divides the work into 13 separate tasks. Example tasks for the benchmark include for example transmit and receive tasks to a base station, communication between the two CPUs using SPI and controlling altitude. It also includes a definition of the hardware as well as interrupts and how often they should trigger. *TACLeBench* is a suite of benchmarks, which also includes *PapaBench*, focused on usage in timing-analysis. *TACLeBench* is used in this paper for evaluation of emulation mode in *gem5*.

Rather than using expensive hardware, another option is to use clock-cycle accurate simulators. *Gem5* is one such product, with research done on its accuracy [7, 8]. Due to being a simple executable, *gem5* is capable of being distributed over many cores. Moreover, *gem5* is very customisable, providing options for simulating different CPU architectures, memory configurations, full L1/L2 cache simulation, power measurements and more.

Given that many programs that require WCET analysis are in the embedded software class and also run on ARM technology, as well as being well-documented and compared to real hardware, the in-order HPI CPU model [9] is most applicable to researching WCET.

Another thing to note is the difference between System Call emulation mode (SE) and Full System emulation mode (FS). In SE mode, all system calls are passed on to the host system. For the HPI CPU model, one of the consequences of SE mode is that I/O does not go over the HPI memory bus and instead gets a simulated delay. Furthermore, as the simulation in SE mode does not include simulating Linux kernel, the interference of CPU scheduler is not taken into account. FS mode on the other hand, includes a root file system with a full Linux kernel and simulations in this mode include interrupts, full I/O over memory bus simulation as well as CPU scheduler and background processes included in all measurements.

Of these two modes, SE mode is the easiest to distribute over a large number of cores, as starting a new simulation is a simple matter of starting a new instance of *gem5* with the proper parameters. FS mode does not necessarily mean that this is not possible, except that feeding the running instances with new programs requires having *gem5* connect via TCP/IP or *gem5*'s readfile/writefile instructions to the outside world. Both of these solutions would influence the simulation itself.

Another limitation of *gem5* is that it only supports several operating systems that it can emulate [10]. For the ARM platform, these are limited to Linux, BSD and Android.

An example of the statistics that *gem5* dumps is available in listing 2.1.

Listing 2.1: Example of some of the statistics that *gem5* dumps

```
final_tick       12680750000 # Number of ticks from beginning of simulation (restored
    from checkpoints and never reset)
host_inst_rate   250793      # Simulator instruction rate (inst/s)
host_mem_usage   407392      # Number of bytes of host memory used
host_op_rate     269462      # Simulator op (including micro ops) rate (op/s)
host_seconds     4.08        # Real time elapsed on the host
host_tick_rate   3069915797  # Simulator tick rate (ticks/s)
sim_freq         1000000000000   # Frequency of simulated ticks
sim_insts        1023936     # Number of instructions simulated
sim_ops          1100184     # Number of ops (including micro ops) simulated
sim_seconds      0.012534    # Number of seconds simulated
sim_ticks        12534240000 # Number of ticks simulated
system.cpu_cluster.cpus.branchPred.lookups  219524  # Number of BP lookups
system.cpu_cluster.cpus.branchPredindirectMispredicted  4   # Number of mispredicted
    indirect branches.
system.cpu_cluster.cpus.committedInsts  1015910 # Number of instructions committed
system.cpu_cluster.cpus.committedOps    1090253 # Number of ops (including micro ops)
    committed
system.cpu_cluster.cpus.cpi 1.233794    # CPI: cycles per instruction
system.cpu_cluster.cpus.dcache.ReadReq_accesses::total  280366  # number of ReadReq
    accesses(hits+misses)
system.mem_ctrls.busUtil    0.04   # Data bus utilization in percentage
```

Gem5 includes detailed statistics on the simulations it runs. Instructions simulated, time simulated, time taken by host, cache misses, page faults, TLB accesses and more are all logged. *Gem5* automatically logs these statistics upon program exit. For SE mode this means a relatively accurate collection of data that the program took to run, given the constraints of its mode. In FS mode however, there is no definitive exit, as it emulates a full Linux system. Hence, *gem5* offers assembly instructions on each CPU model, that tell *gem5* to reset counting and dumping of statistics. These instructions are exposed as C functions in a header, but that automatically means that the software under test has to be modified explicitly to tell *gem5* which moments statistics should be collected. As can be seen in listing 2.2, the modification for dump statistics is a minor one. For FS mode however, executing a separate binary with these instructions exposed as command line arguments is available. A downside of this is that this executable has to be loaded from disk into memory, adding overhead to the statistics and filling the CPU cache with its execution. This overhead has been measured on our setup, as described in Chapter 3, to be 548.6 μs(n=10, σ=32.2).

Listing 2.2: Example of how to dump stats with *gem5* by modifying source code

```c
#include "gem5/m5ops.h"

// lift_init, lift_main and lift_return functions left out for brevity

/* m5_dump_stats(0, 0) results in the following assembly on AArch64:
 * 400774:       d2800001        mov     x1, #0x0                        // #0
 * 400778:       d2800000        mov     x0, #0x0                        // #0
 * 40077c:       94000042        bl      400884 <m5_dump_stats>
 */

int main( void )
{
  lift_init();

  m5_dump_stats(0, 0);
  lift_main();
  m5_dump_stats(0, 0);

  return ( lift_return() );
}
```

2.3 Edify

Edify [3] laid some of the groundwork for this paper. *Edify* works by first using a static analyser to classify variables along two orthogonal lines: directly or indirectly influencing execution times and whether the influence is on loops, conditionals or variable instruction times. This output can then be used to reduce the set of variables to iterate over, reducing the state-space. The *Edify* framework then runs a number of simulations until a fixed-point iteration is achieved. This then results in an execution time distribution. However, *Edify* focuses only on single-core executions and only on the values of variables. There are more factors influencing execution time, such as cache state, interrupts and in the case of multi-core, the time when a task on core 1 executes relative to a task on core 2.

As the static analyser seems unwieldy to use and incomplete, an alternative would be preferred. In companies applying WCET analysis, the source code is accessible and someone is available who helped create the software, which means that the analyser step can be replaced by manually selecting variables that influence execution times. As such, this static analyser is not present in our framework and we will manually determine what the output of this tool would have given us.

Chapter 3

Accuracy of SE vs FS mode

As mentioned, *gem5* offers two simulation modes. Even though others [7, 8] have done measurements on the accuracy of *gem5*, both cited papers use the FS mode to do the measurements and have not compared accuracy between the two modes. The HPI model [9] does measure SE and FS performance individually, but does not make an attempt at comparing the two. Moreover, accuracy of the simulations also depends on single-core vs multi-core. In addition, it is easier to distribute *gem5* over multiple processors. As such, a closer view on whether SE mode is accurate enough for our state-space analysis is required.

TACLeBench provides benchmarks for both modes and has been selected for comparing the two modes using the HPI CPU model, the default DDR3 1600 8x8 1 memory channel with 256 MB, L1 cache enabled and for multi core L2 enabled and the CPU frequency set to 100 MHz. Since *gem5* only shows statistics with an accuracy up to 1 microsecond, running *gem5* at the default 2 GHz would result in certain benchmarks completing in 0 μs. The software has been modified slightly, for each benchmark we inserted a function call to tell *gem5* to dump statistics after the initialisation and after the main task. With this we can measure the duration of the task itself.

For the single-core benchmarks, Table 3.1 shows a single full run of *TACLeBench* on SE and FS compared between each other. Aside from some anomalies, simulated time is very close between the two modes. Two benchmarks from the sequential directory have been excluded (*rijndael_enc* and *rijndael_dec*) since they created a "stack smashing detected" error, a protection added in recent gcc releases. The *fft*, *lms* and *h264_dec* benchmarks produce incorrect results on our setup. On x86-64 the *fft* and *h264_dec* benchmarks do produce correct results, but even on x86-32 the *lms* benchmark produces incorrect results. Moreover, the recent gcc release used in our setup catches an undefined behaviour bug in the *lms* benchmark. The bug in *lms* has been reported by us and our fix has been merged after we finished our benchmarks, but the other bugs are reported and we have not heard anything back on those. Regardless, this suggests that *TACLeBench* might not be well tested on the AArch64 and x86-32 architectures.

The big outlier is the *fac* benchmark. In the main function of that benchmark, printf is called. Given that we ran a Linux distribution using glibc, printf both allocates memory and obtains a lock, both of which are system calls. Removing the printf statement makes the simulated time in FS mode equal to SE mode, whereas doubling the printf call only adds half of the simulated time, suggesting that some caching effect is going on. This is most likely the reason why a benchmark like *huff_enc* is not affected as badly, given it does 648 printf calls in the main function. In a scenario which uses system calls, it might be better to use FS mode, but system calls are generally expensive and thus they tend to be avoided in embedded software or in the case of running without an OS they are not present at all. This is also the case for *PapaBench*, which does not impede the accuracy for SE mode significantly in our context.

The benchmarks *ndes*, *petrinet* and *statemate* show the biggest relative differences. *Petrinet*, and other benchmarks that finish within roughly 20 μs, tend to skew the overall difference. A 4 μs difference for *petrinet* means 57.1%, whereas a 4 μs extra difference for *ndes* would mean less than a percentage-point difference. Therefore, it is much more interesting to look at the *ndes* and *statemate* benchmarks. If we compare the statistics *gem5* gives us for SE and FS mode for both *ndes* and *statemate* we find that the amount of instructions committed in FS mode is significantly more than SE mode. Moreover, the cycles per instruction count is higher as well, though *statemate* shows a lesser increase for this statistic than *ndes*. This suggests two things: the former difference suggests that in FS mode, more than just the benchmark is executed. This is a given, since FS mode includes things like CPU schedulers and background processes which may execute during the benchmark. The latter suggests that things like the branch predictor and the L1 cache have a lower hit-rate, most likely also because of background processes filling the stored branch predictions and caches with its own data simply by being executed. Although we have no definitive evidence that these factors are the cause of these differences, they are likely enough to leave determining the exact impact for future work.

Interesting to note about these single-core benchmarks is that the longer the benchmarks run, generally speaking, the less difference there is between the two modes. Figure 3.1 shows this more clearly. This suggests that part of the difference between SE and FS can be explained by some overhead in FS mode: the very fast benchmarks sometimes complete in (almost) the same time, sometimes in a lot slower time, but as the length of the benchmark increases, the chance of running into this overhead increases to 100% but the impact decreases. This lowers the spread of the benchmarks in the bottom-right quadrant. We think this is most likely because of extra processes running in the background in FS mode, such as loggers and system daemons.

We also tried to run the multi-core benchmarks, namely the *DEBIE*, *PapaBench* and *rosace* benchmarks included in *TACLeBench*. *DEBIE* has been modified by the authors of *TACLeBench* to be single core, *PapaBench* is a two-core benchmark and rosace a five-core benchmark. However, as *gem5* in FS mode with 3 or more CPU cores crashed very early in the boot process, and we already had single core results, we could only meaningfully run *PapaBench*. Making measurements in FS mode with multiple cores involves more than measurements for single core: each executable has to be started separately at roughly the same time, but starting executables include simulating I/O, which makes a second started executable always start later. It is also very hard to determine which statistics belong to which *m5_dump_stats* call, given that they can be interleaved. If the executables are especially fast, it is likely that the first executable finishes before the second executable. Fortunately, *gem5* supports setting CPU frequency, reducing the speed with which *PapaBench* completes. We inserted a call to *gem5* to dump stats before and after the main part of the benchmark, for both executables, excluding initialisations. This results in four calls to dump statistics, of which the difference between the first and fourth call are then considered to be the execution time for FS mode. We ran *PapaBench* with the CPU frequency set to 100 MHz and 1000 MHz. The results can be seen in Table 3.2.

Given that *PapaBench* aims to emulate an embedded system with a processor core dedicated to one executable, the setup that *gem5* gives us seems to be outside of the intended use. There is likely no CPU scheduler in such a setup, disk I/O is only present during system boot for loading the executables into RAM, the executables also start more deterministically in such a system. On top of that, running Linux means a whole combination of things: no ring 0 access, using entire software stacks such as the GNU C library, aforementioned extra processes running in the background such as loggers and system daemons and extra security measures such as stack smashing protection or SPECTRE mitigations. Although the extra security measures can be disabled, it is not the default. For single-core, we can reasonably show that the choice between SE and FS does not impact the execution time of the tasks a lot, outside of some certain known situations. For multi-core however, there a lot of setup differences that make comparing SE and FS mode difficult. The per-run variation that FS mode introduces due to the aforementioned problems for both single-core and multi-core simulations, makes isolating factors that impact execution times a lot harder. Even with this difficulty, we think it is likely that the difference for *PapaBench* between SE and FS mode can be explained by other Linux processes running in the background as well as disk I/O having an impact on when the second *PapaBench* executable starts. Though validation hereof through synchronisation of executable start times is left to future work. These differences combined with the points that it is quite unlike embedded software systems to run Linux instead of a real-time operating system or no operating system, and that it is easier to distribute SE mode over multiple processors, we have opted to go with SE mode for our framework.

Table 3.1: SE vs FS single-core results.

Benchmark	Simulated time SE (μs)	Simulated time FS (μs)	FS difference with SE
fac	27	945	3,500%
ndes	666	1,194	79.2%
petrinet	7	11	57.1%
statemate	515	731	41.4%
prime	3	4	33.3%
duff	3	4	33.3%
adpcm_dec	16	21	31.2%
h264_dec	177	224	26.6%
audiobeam	2,480	3,063	23.5%
huff_dec	1,310	1,598	22.0%
huff_enc	4,019	4,901	21.9%
anagram	12,949	15,570	20.2%
adpcm_enc	20	24	20.0%
fmref	3,219	3,862	20.0%
fft	3,093	3,677	18.9%
insertsort	6	7	16.7%
powerwindow	12,522	14,598	16.6%
minver	19	22	15.8%
cjpeg_wrbmp	505	584	15.6%
sha	7,481	8,619	15.2%
g723_enc	3,994	4,581	14.7%
gsm_dec	10,844	12,446	14.7%
gsm_enc	21,374	24,011	12.3%
cubic	8,837	9,906	12.1%
lift	4,768	5,331	11.8%
epic	22,205	24,746	11.4%
isqrt	4,544	5,050	11.1%
cjpeg_transupp	16,975	18,865	11.1%
ludcmp	19	21	10.5%
filterbank	24,521	26,770	9.2%
jfdctint	11	12	9.1%
quicksort	15,492	16,782	8.3%
cover	12	13	8.3%
test3	849,804	920,944	8.3%
fir2dim	25	27	8.0%
pm	71,651	76,260	6.4%
md5	85,456	90,322	5.7%
susan	240,178	252,808	5.3%
dijkstra	217,516	228,557	5.1%
recursion	21	22	4.8%
ammunition	742,704	778,179	4.8%
mpeg2	1,067,655	1,118,114	4.7%
bitcount	66	69	4.5%
countnegative	29	30	3.4%
bitonic	98	100	2.0%
cosf	170	172	1.2%
bsort	530	531	0.2%
binarysearch	1	1	0%
complex_updates	4	4	0%
deg2rad	55	55	0%
iir	2	2	0%
lms	1,107	1,107	0%
matrix1	65	65	0%
rad2deg	55	55	0%
st	415	415	0%
total (55 benchmarks)	3,460,240	3,676,032	6.2%

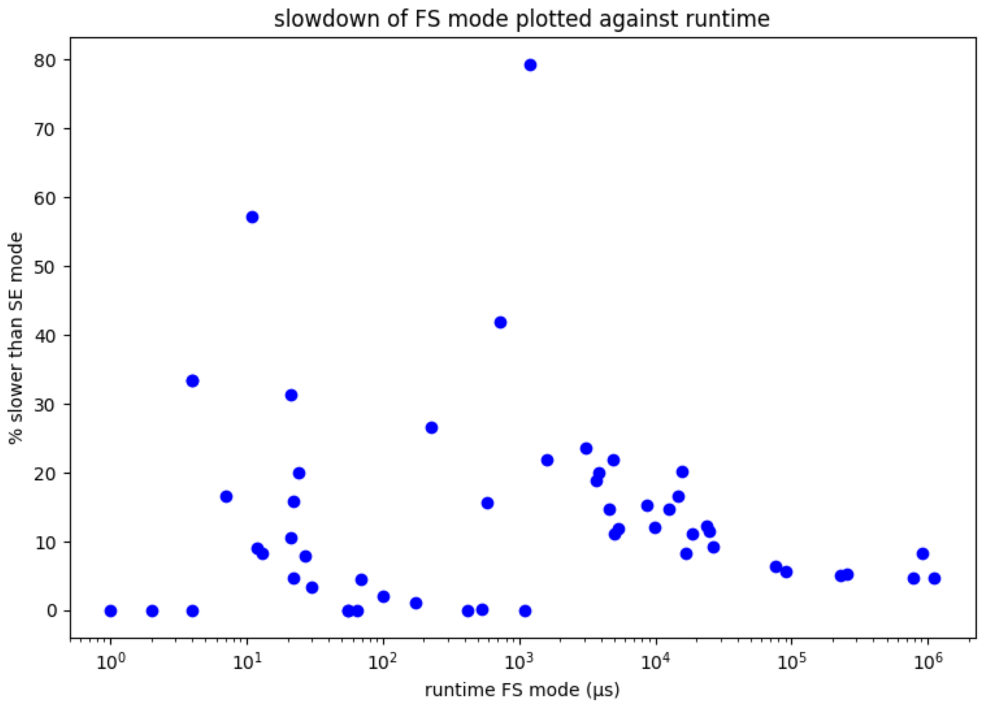


Figure 3.1: Difference between FS and SE mode plotted against runtime, excluding fac.

Table 3.2: SE vs FS *PapaBench* multi-core results at 100 MHz and 1000 MHz CPU Frequency, standard PapaBench build.

CPU Frequency (MHz)	Run number	Time SE (μs)	Time FS (μs)	Difference
100	1	580	2742	472.8%
100	2	580	4531	781.2%
100	3	580	3799	655.0%
100	4	580	3712	640.0%
100	5	580	4927	849.5%
1000	1	69	183	265.2%
1000	2	69	403	584.1%
1000	3	69	1016	1472.5%
1000	4	69	566	820.3%
1000	5	69	571	827.5%

Chapter 4

State-space framework analysis

PapaBench consists of 8 tasks for the *autopilot* program and 5 tasks for the *flybywire* program. Table 4.1 lists the number of branches, and thus required simulation runs, required for brute forcing WCET for each individual task. For the default *PapaBench* test benchmark, the tasks are always executed in the same order, but the original paper for *PapaBench* [2] shows that the intention is to run each task a set amount of times per second and includes interrupts. *Gem5* does not support setting up interrupts, so we are unable to include those in our analysis. We will instead focus on shuffling of task order and variables influencing execution times. Because of time pressure, the constraint of running certain tasks after other tasks is not met, leading to configurations that would normally result in wrong state. This leads to tasks being run at the same time which would not happen in reality, possibly leading to a more optimistic or a more pessimistic execution time. Moreover, the task initialisation only initialises the variables that directly influence execution times of the task itself. However, some tasks, such as the *altitude_control_task* task, influence the content of variables used in other tasks, such as *climb_control_task*. In this case, the variable *desired_climb* is set by *altitude_control_task* and used as an input for *climb_control_task* with several if statements influenced by this value. With the way that SE mode works in *gem5*, each executable operates with its own memory range, preventing modifying contents of memory of tasks on one core by tasks of the other core. This means that in the current setup, all tasks scheduled to run on one core, can have an impact on variables of other tasks running on that core but not the other core. Preventing this effect on variables used by other tasks is possible by setting them to a value in the task initialisation. This effectively treats the variables influencing other tasks as variables influencing execution times, increasing the state-space through the amount of branches taken. However, the impact of this decision is unknown to us: it could be an important effect that task order can have on execution times, but it can also make it harder to isolate cause and effect. We have opted to allow these cross-task influences to occur to reduce the amount of simulations required and studying the impact is left for future work.

4.1 State-space framework

To reduce the state-space of multi-core simulations, we want to see if cutting separate execution time influencing factors into separate steps is an option. For testing this, we have selected the variables influencing execution times, like *Edify* has shown, and the order in which tasks are executed. More factors, such as thread delay or cache state, would give a less optimistic result for WCET, but we are interested only in seeing if handling each factor separately, using the output of one step as input to the other step, is an option that does not compromise on accuracy too much.

We have implemented a framework that allows us to run simulations on the *DAS-4* [11, 12] supercomputer using *MPI* and *gem5*. *DAS-4* is a slightly older supercomputer being in operation since 2010, with 2 clusters already being decommissioned to make room for *DAS-5*. Our research used a cluster that supported executing on 32 nodes simultaneously, resulting in an availability of 256 CPUs. An overview of the framework is as follows (see also figures 4.1 and 4.2): the software to be simulated is modified to expose itself to our framework, see Section 4.2 for more detailed information, and a configuration file given the tasks and variable iteration range is made. The software to be simulated then offers an interface to our framework to execute threads in arbitrary order as well as with which values the tasks are to be initialised.

The framework is run with *MPI*, each node starts up a python script like what is shown in listing 4.4. For determining values, a slightly different python script is defined, which does not permute over task order but creates a Cartesian product of all possible values defined in the configuration instead. Lines 21 through 23 of listing 4.4 are the only interaction with MPI, as there is no need for communication between MPI nodes. Instead, line 4 shows that the slice of permutation is computed for each node, depending on its own node number and the total number of nodes that are running the framework: rank and size respectively. Calculating all permutations and distributing them over MPI would require orders of magnitude more RAM, which would quickly become more than what the master node could allocate.

To reduce the number of simulations, our framework uses a preliminary step: calculate the values that lead to the WCET when running tasks separately. Our assumption is that the simulation with the highest number of instructions simulated is the slowest path through the task. Given that another assumption is that programs used with our framework can only be of the type where the user has complete control over the source code and has knowledge of the software, it is likely that users can select interesting variables. Using the executable made when modifying the software to be simulated, our framework then simulates all possible paths given the selected values. Therefore, only permuting over task order remains, significantly reducing the number of simulations needed. Our setup permutes over task order indiscriminately, that is, task dependencies are ignored, as explained in the intro of this chapter.

For each permutation of task order, the tasks are distributed over each core of the two cores. This means that, for the 13 tasks of *PapaBench*, 14 simulations are run. For example, if a permutation results in the following task order: $\{1, 2, 3, 4, 5, 13, 12, 11, 10, 9, 8, 7, 6\}$ the following simulations are run, with each embedded set being the simulations for one core:

$$\{\{\emptyset\}, \{1, 2, 3, 4, 5, 13, 12, 11, 10, 9, 8, 7, 6\}\}$$

$$\{\{1\}, \{2, 3, 4, 5, 13, 12, 11, 10, 9, 8, 7, 6\}\}$$

$$\{\{1, 2\}, \{3, 4, 5, 13, 12, 11, 10, 9, 8, 7, 6\}\}$$

and so on up to

$$\{\{1, 2, 3, 4, 5, 13, 12, 11, 10, 9, 8, 7, 6\}, \{\emptyset\}\}$$

Once either the simulations are completed or the supercomputer job time limit has been reached, all statistics are collected and put into an aggregation tool to create an execution time distribution. The number of simulations necessary to simulate each branch combination for *PapaBench* would have lead to $\prod_{i=1}^{13} p_{t_i} = 1.208 \times 10^{14}$ (Table 4.1, p = paths of task i) combinations, but only wanting the highest execution time per task needs $\sum_{n=1}^{13} p_{t_i} = 1,396$ simulations. Permuting over task order and spreading them over both cores leads to $\frac{13!}{(13-13)!} \times (13+1) = (13+1)! = 8.718 \times 10^{10}$ combinations.

The statistics dumped by *gem5* are compressed using Zstandard [13], which is one of the best compression techniques when it comes to high compression and decompression throughput. Zstandard is the leading-edge in free compression techniques that outperforms other techniques for its intended compression ratio range. Each simulation generates about 4,5 MB worth of statistics. Using Zstandard with a trained dictionary, this is reduced to 40-50 KB. Since our allotted disk quota is 40 GB, this limits us to roughly 700,000 simulations before we run out of space. A variation of other free compression techniques, namely 7zip, bz2, gz2 and zip, have been tried, of which only 7zip was able to improve on compression ratio while having significantly lower compression throughput.

To assess the accuracy of dividing the state-space into two steps, we define the following runs:

- one run to determine the values of the variables influencing execution times, isolated per task. This run does not permute over task order.
- One run using the values found in the previous run as input, only permuting over task order.

- One validation run with the task order with the highest execution time found in the previous run, fixing the task set order and distribution over cores, and iterate over the Cartesian product of all possible values of variables influencing execution time. In contrary to the first run, this is not isolated per task.

In the last run, the aim is to find any simulations which lead to a higher execution time than what was found in the second run. If there are none, we can say with a high likelihood that reducing state-space in this manner does not lead to lower accuracy.

Finally, we compare a random simulation from the second run with a hand-made binary, to determine the overhead of the API we introduce.

Table 4.1: Tasks and identified unique paths of *PapaBench*.

Task	Paths
T1: check_failsafe_task	2
T2: check_mega128_values_task	16
T3: send_data_to_autopilot_task	16
T4: servo_transmit	1
T5: test_ppm_task	16
T6: radio_control_task	320
T7: stabilisation_task	2
T8: link_fbw_send	3
T9: receive_gps_data_task	192
T10: navigation_task	25
T11: altitude_control_task	2
T12: climb_control_task	800
T13: reporting_task	1

4.2 Benchmark software setup

The framework requires manual setup on the software to be simulated beforehand. This includes dividing the software into tasks, manually determining code paths per task, altering the source code to introduce an initialisation function per task which can be used by *Edify* to trigger all code paths of that particular task, introducing a function that runs a task as well as modifying the output to be a library rather than an executable. For *PapaBench*, the tasks have already been clearly divided and the benchmark contains several variables that influence execution time. As explained in Section 2.3, we looked at the source code ourselves and identified several more code variables that directly influence execution times, the result of which can be seen in Table 4.1.

These modifications are applied on the source code under test, which has to expose a linked list with all the tasks, and per task: its name, initialisation function and the run function. Listing 4.1 shows an example of this and listing 4.2 shows an example of how a task is initialised.

A configuration file is then used to determine what values should be injected into the tasks, matching the tasks by order of appearance and matching the values per task also by order of appearance. Listing 4.3 demonstrates how this could look.

Once setup is complete, an executable is created combined with the aforementioned library. This executable offers a command line arguments interface for our framework to decide which tasks to run, in which order and with which initialisation values. An example of how the executable is called from the command line is available in listing 4.5. This example executes task *check_mega128_values_task* first with the values 0, 1, 0 and 1 given to the initialise function, then the task *send_data_to_autopilot_task* with the values 1, 1, 1 and 1 and lastly the task *link_fbw_send* with the value 2. The order of the values given on the command line correspond to the order of the values defined in the initialisation of the task, like in listing 4.2. The names of values in the configuration file serve the purpose of improving readability.

Listing 4.1: Example of exposing two tasks and the program initialisation for *PapaBench*

```c
void program_init(void);

void servo_transmit(void);

void test_ppm_task(void);
int init_test_ppm_task(int arg_count, int *task_args);

typedef void (*functionPtr)();
typedef int (*initFunctionPtr)(int count, int *task_args);

typedef struct task {
    char *name;
    functionPtr function;
    initFunctionPtr init;
    struct task* next_task;
} task;

task _servo_transmit = { .name = "servo_transmit", .function = &servo_transmit, .init
    = 0, .next_task = 0 };
task _test_ppm_task = { .name = "test_ppm_task", .function = &test_ppm_task, .init = &
    init_test_ppm_task, .next_task = &_servo_transmit };
task tasks_to_execute = { .name = "program_init", .function = &program_init, .init =
    0, .next_task = &_test_ppm_task };
```

Listing 4.2: Example of initialisation of global variables for the test_ppm_task task

```c
int init_test_ppm_task(int arg_count, int *task_args) {
    if(arg_count != 4) {
        printf("Horrible disaster for init_test_ppm_task, expected 4 arguments got %i\
            n", arg_count);
        return -1;
    }

    mode = task_args[0];
    ppm_valid = task_args[1] == 1 ? TRUE : FALSE;
    spi_was_interrupted = task_args[2] == 1 ? TRUE : FALSE;
    last_radio_contains_avg_channels = task_args[3] == 1 ? TRUE : FALSE;

    return 0;
}
```

Listing 4.3: Part of the configuration file used for step 1 in our framework: finding the values which lead to the highest execution time

```json
{
  "papabench": {
    "tasks": [{
      "name": "send_data_to_autopilot_task",
      "values": [{
        "name": "mode",
        "values": [0, 1]
      },{
        "name": "SPI_PIN",
        "values": [0, 1]
      },{
        "name": "spi_was_interrupted",
        "values": [0, 1]
      },{
        "name": "last_radio_contains_avg_channels",
        "values": [0, 1]
      }]
    },{
      "name": "check_mega128_values_task",
      "values": [{
        "name": "mode",
        "values": [0, 1]
      },{
        "name": "SPI_PIN",
        "values": [0, 1]
      },{
        "name": "spi_was_interrupted",
        "values": [0, 1]
      },{
        "name": "last_radio_contains_avg_channels",
        "values": [0, 1]
      }]
    },{
      "name": "link_fbw_send",
      "values": [{
        "name": "spi_cur_slave",
        "values": [0, 1, 2]
      }]
    }]
  }
}
```

Listing 4.4: MPI distribution of permuting over task order

```python
# Imports, some functions, classes and arguments skipped for brevity
class GenerateThreadsSimulationsTask:
    def produce_task_permutations(self, tasks: List[Task]) -> Iterator[List[Task]]:
        for workload in itertools.islice(itertools.permutations(tasks, len(tasks)), int(self.rank * self.skip), int((self.rank + 1) * self.skip)):
            yield workload

    def execute(self):
        run_id = 1
        for task_permutation in self.produce_task_permutations(self.benchmark.tasks):
            for x in range(len(self.benchmark.tasks) + 1):
                core_one: List[Task] = task_permutation[:x]
                core_two: List[Task] = task_permutation[x:]
                run_args: List[str] = [self.get_run_args(core_one), self.get_run_args(core_two)] #create command line parameters per core

                RunSimulationTask(run_args, self.rank, run_id).execute() #this runs gem5, compresses stats, stores results on the filesystem and so on

                run_id += 1

if __name__ == "__main__":
    benchmark_config = BenchmarkConfig(json.load(open('benchmarks.json')))
    comm = MPI.COMM_WORLD
    size = comm.Get_size()
    rank = comm.Get_rank()
    benchmark, = [bench for bench in benchmark_config.benchmarks if bench.name == "papabench"]
    total_permutations = math.factorial(len(benchmark.tasks)) #not the same as calculated state-space! Distributing over cores is listed above
    skip = int(total_permutations/size)

    GenerateThreadsSimulationsTask(benchmark, rank, skip).execute()
```

Listing 4.5: Example usage of *PapaBench* executable with command line API

```
./papabench check_mega128_values_task;send_data_to_autopilot_task;link_fbw_send 0;1;0;1 1;1;1;1 2
```

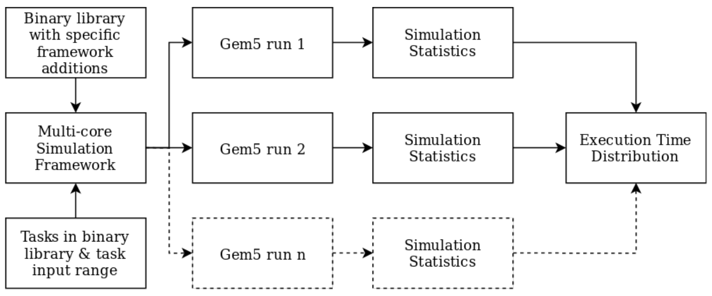

Binary library with specific framework additions
Multi-core Simulation Framework
Tasks in binary library & task input range
Gem5 run 1
Gem5 run 2
Gem5 run n
Simulation Statistics
Simulation Statistics
Simulation Statistics
Execution Time Distribution

Figure 4.1: Overview of multi-core WCET framework: the benchmark configuration (bottom left) and executable with command line API (top left) serve as input; our framework is then distributed over all nodes through MPI, resulting in a collection of statistics from which an execution time distribution can be extracted.

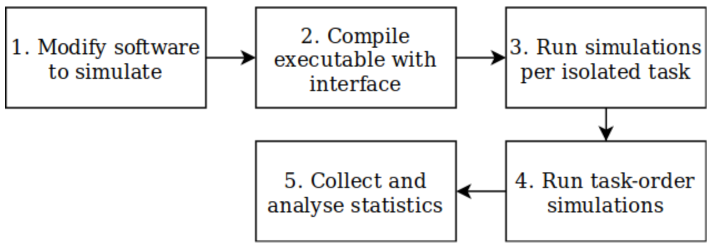

1. Modify software to simulate
2. Compile executable with interface
3. Run simulations per isolated task
4. Run task-order simulations
5. Collect and analyse statistics

Figure 4.2: Stages of multi-core WCET framework: how we arrive at an execution time distribution.

Chapter 5

Measurements using framework for *PapaBench*

Table 5.1: Results using worst-case values after running X minutes of simulations, full task set of PapaBench with 13 tasks.

Time (minutes)	Min (μs)	Max (μs)	Total Simulations	Incorrect Simulations
15	177	623	39,130	93
30	177	623	82,780	211
60	158	623	169,922	449
90	158	623	256,721	688
120	150	624	343,601	913
240	150	624	690,777	1973

Table 5.2: Results using worst-case values, reduced task set of PapaBench with 7 tasks.

Main or Validation run	Min (μs)	Max (μs)	Total Simulations	Incorrect Simulations
Main	41	115	40,320	57
Validation	48	148	98,304	0

Figure 5.1 shows the execution distributions of all permutations of the 13 tasks in *PapaBench*, using the values for variables leading to the highest instruction count per task. Each graph represents the distribution found after the given minutes of running simulations. The final, 240 minute graph shows the distribution for the entirety of our dataset. Table 5.1 shows the minimum and maximum simulation time found by summing the execution times of all tasks, as well as the total simulations and the number of incorrect findings, for their respective time frame. A fixed-point iteration is found at the 120-minute mark, after which no new execution times are found when compared to the 240-minute run. Our framework ran an average of 2878 simulations per minute on 256 cores, slightly more than 11 simulations per core per minute. The incorrect simulations are due to two categories of errors. Category one is Zstandard checksum errors and incorrect amount of dumped stats. This category is likely due to storing the output on the distributed filesystem led to bytes being dropped. We tried to reproduce the incorrect amount of dumped stats with exactly the same setup as during the full run on a different machine, but we could not manage to do so. Category two is missing simulations due to being aborted by the job system. The files were already created but no data written to them. These are all also the last run a particular node has made. Category 1 happens 1926 times out of the 1973 incorrect simulations noted, only 1 is due to checksum errors and the remaining 46 are due to being aborted.

To determine whether reducing state-space by two steps gives optimistic or pessimistic execution times, we take the task order of one of the runs that has the highest execution time from our dataset

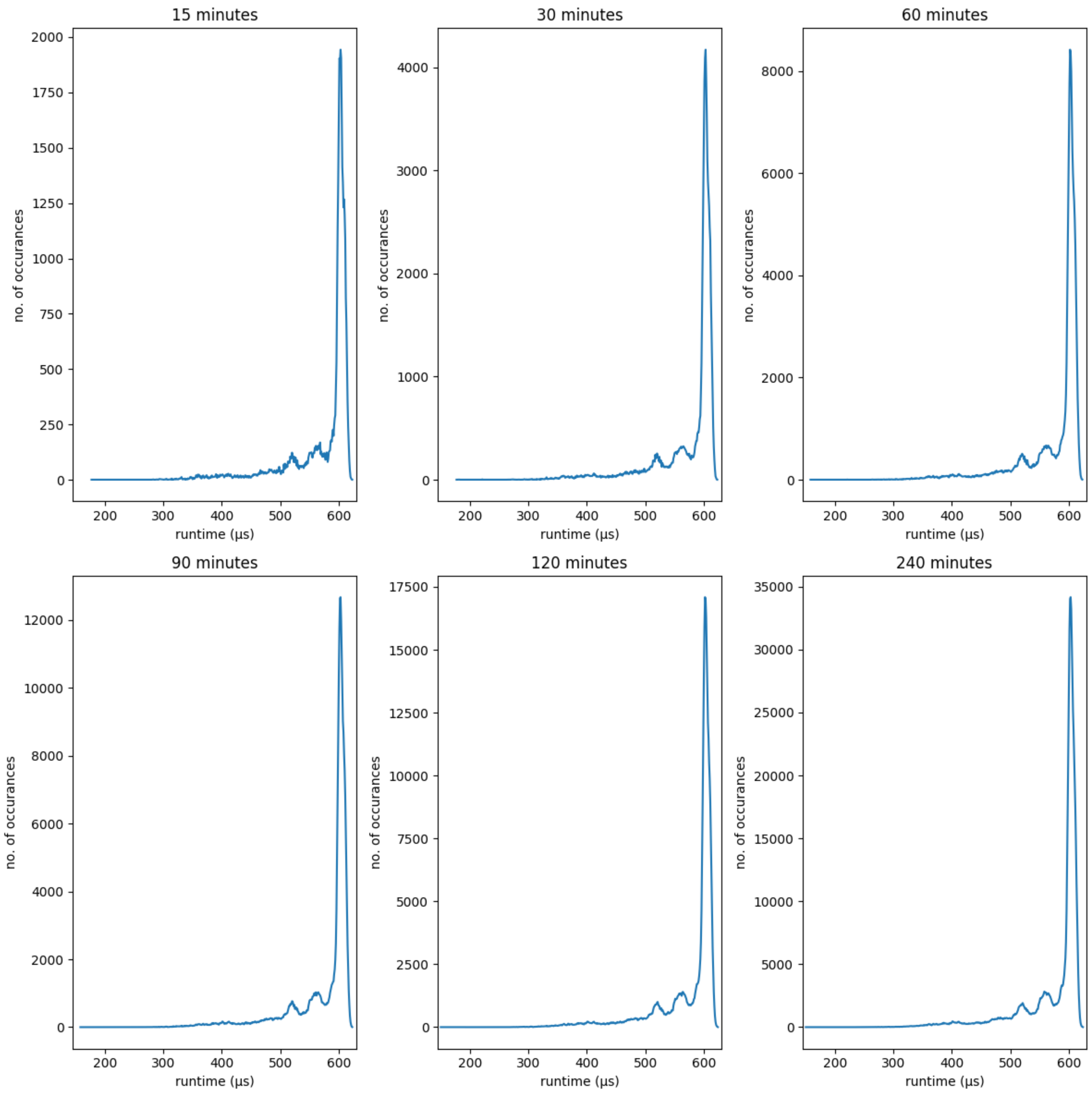
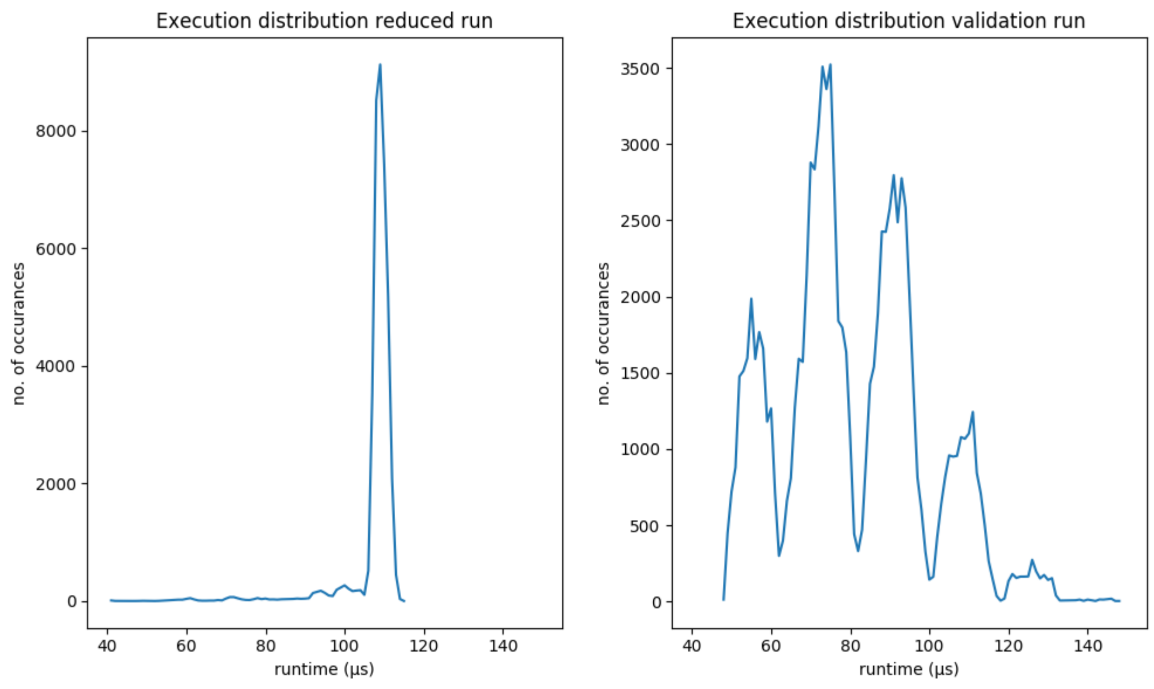


Figure 5.1: Execution distribution of *PapaBench* at various times. Each subgraph shows how often a specific runtime was encountered, within X minutes of starting the state-space analysis framework. The area under the graphs are the same as the total number of simulations in Table 5.1 minus the incorrect simulations.



(a) Main run

(b) Validation run

Figure 5.2: Execution distribution of smaller run of *PapaBench*. Each subgraph shows how often a specific run-time was encountered. The area under the graphs are the same as the total number of simulations in Table 5.2 minus the incorrect simulations. The main run here is permuting over task order, the validation run simulates all the possible values of variables directly influencing execution time, with the same task order as the simulation with the highest run-time found in the main run.

of 240-minutes, and simulate all combinations of values of variables identified in Table 4.1, with this task order. We call this our validation run. A validation run on *PapaBench* with 13 tasks requires 1.208×10^{14} simulations, which would take almost 80,000 years on our setup. We have therefore also included a smaller run in our experiments, with only 7 of the 13 tasks of *PapaBench*: T1, T2, T3, T5, T7, T8 and T11. This leads to needing $2 \times 16 \times 16 \times 16 \times 2 \times 3 \times 2 = 98,304$ simulations for validation and $(7+1)! = 40,320$ permutations of task order. The results of the permutation of tasks run as well as the validation run can be seen in Table 5.2. figures 5.2a and 5.2b show the execution distributions of the runs, respectively. The validation run shows that there are quite some configurations where the execution time is higher than the highest configuration from the main run. Moreover, it seems that modifying variables that influence execution times leads to greater diversity in the distribution, suggesting that it is a bigger factor than task order.

Comparing a random simulation from the full *PapaBench* task set run with a binary made with a static task order execution, without the command line API, we find two things: the measured overhead in terms of instructions from our framework is only a pointer indirection, that is 2-4 instructions per task depending on how the compiler arranges the instructions. Second, for the run we compared, the hand-made binary takes 11 μs more to complete but 24 fewer instructions. *gem5* outputs that the cycles per instruction for the hand-made binary is significantly higher than the run with our framework as well as a lower success rate for the branch predictor, however the total number of branch predictor lookups done is lower in the hand-made binary. Cache hit/miss statistics show no significant differences between the runs.

Listing 5.1: Static task order binary, used to compare with our *PapaBench* executable with command line API, to determine overhead of our framework

```c
void test_ppm_task(void);
int init_test_ppm_task(int arg_count, int *task_args);

// SNIP
// more extern definitions of functions here
// SNIP

#define EXECUTE_TASK_WITH_INIT(task, valcount, valone, valtwo, valthree, valfour,
    valfive, valsix) \
    values[0] = valone; values[1] = valtwo; values[2] = valthree; values[3] = valfour;
         values[4] = valfive; values[5] = valsix; \
    init_ ## task(valcount, values); \
    m5_dump_stats(0, 0); task(); m5_dump_stats(0, 0); \

#define EXECUTE_TASK(task) \
    m5_dump_stats(0, 0); task(); m5_dump_stats(0, 0);

int main( int argc, char *argv[] ) {
    int values[6];

    program_init();

    EXECUTE_TASK_WITH_INIT(test_ppm_task, 4, 1, 0, 1, 1, 0, 0)
    EXECUTE_TASK(servo_transmit)
    EXECUTE_TASK_WITH_INIT(send_data_to_autopilot_task, 4, 0, 1, 1, 1, 0, 0)
    EXECUTE_TASK_WITH_INIT(check_mega128_values_task, 4, 0, 1, 0, 1, 0, 0)
    EXECUTE_TASK_WITH_INIT(check_failsafe_task, 1, 0, 0, 0, 0, 0, 0)
    EXECUTE_TASK_WITH_INIT(stabilisation_task, 1, 1, 0, 0, 0, 0, 0)
    EXECUTE_TASK(reporting_task)
    EXECUTE_TASK_WITH_INIT(receive_gps_data_task, 6, 0, 3, 0, 0, 0, 6)
    EXECUTE_TASK_WITH_INIT(radio_control_task, 5, 1, 4, 1, 1, 0, 0)
    EXECUTE_TASK_WITH_INIT(navigation_task, 2, 0, 3, 0, 0, 0, 0)
    EXECUTE_TASK_WITH_INIT(link_fbw_send, 1, 1, 0, 0, 0, 0, 0)
    EXECUTE_TASK_WITH_INIT(climb_control_task, 5, 0, 1, 7, 6, 1, 0)
    EXECUTE_TASK_WITH_INIT(altitude_control_task, 1, 1, 0, 0, 0, 0, 0)

    return 0;
}
```

Chapter 6

Discussion

6.1 Task scheduling

Two big things our framework did not implement are task dependencies and scheduling the start of tasks relative to another task. To properly run these task dependencies, either a task scheduler should be implemented or the next task should wait upon the dependent upon task. Moreover, although the overhead of the measured part of the overhead of our API is negligible, the time it takes between tasks is on the same order of magnitude as the *PapaBench* tasks itself. This means that there might be some overlap between running a *PapaBench* task on one core and our task-decision algorithm on the other core. A synchronisation mechanism for starting threads on cores at the same time would be preferable.

Furthermore, as stated in the beginning of Chapter 4, there exists cross-task variable content influence in *PapaBench*. An extension to Edify's [3] static analyser that detects variables influencing execution times, that detects these influences would be helpful in isolating its effects. It is possible that the large difference between minimum and maximum times found in our results are because of these cross-task influences combined with simulating task orders that do not show up in normal usage of *PapaBench*, though our work gives no indication on whether this is the case or not.

On top of that, as *PapaBench* is a software suite for only two cores, all our results only hold in the case of a dual-core system.

6.2 Answering Research Questions

- RQ1: How to tackle the state-space problem?

The results show that while an approach is possible to reduce the state-space, the trade-off is that accuracy is lowered. In our reduced task set setup, we ran a total of $98,304 + 40,320 = 138,624$ simulations, where the full state-space would otherwise be $98,304 \times 40,320 = 3.964 \times 10^9$ simulations. However, the highest execution time found in the main run was 115 μs, 22.3% lower than the highest time, 148 μs, found in the validation run. And it is possible that the validation run is not the highest execution time in our setup.

Moreover, even using this approach, we run into the issue of not being able to run enough simulations to cover the entirety of *PapaBench*, as we had to reduce the task set amount from 13 to 7 with selectively removing tasks that had a high amount of branches. Considering that our framework operated on the DAS-4 supercomputer, which uses old hardware by today's standard, as well as being constrained to a maximum of 40 GB of data, it is quite likely that a stronger computer or collection of computers on a cloud platform would lead to much more results per minute than what we attained. Furthermore, we believe a further look into performance optimisations may be possible. Unfortunately, even if this would lead to a 2x speedup for simulations per core and if it were possible to run it at a linearly scaled fashion, which is unlikely, to 100,000 cores, validating the non-reduced

PapaBench set would still require

$$\frac{TotalSimulations}{simulations/core/min \times speedup \times cores} = \frac{1.208 \times 10^{14}}{11 \times 2 \times 100,000} = 54907252.4\ minutes$$

or almost 105 full years of non-stop simulating. And this is without extra execution time influencing factors such as cache state and interrupts taken into consideration. Simulating the full state-space of the reduced *PapaBench* set without separating it into separate steps, 3.964×10^9 simulations as mentioned above, would cost about 30 hours in this hypothetical case, but more than 2.5 years on DAS-4. It still seems that it is infeasible to approach multi-core execution time distribution in this fashion.

Whether the reduced accuracy of the execution time distribution is acceptable or not is unknown, as we do not know of any execution time results of *PapaBench*.

- RQ2: Which emulation mode of *gem5* can best be used in large state-space settings?

As we have demonstrated in Section 3, for the niche of embedded software that runs on real-time operating systems or without an operating system, combined with running a lot of simulations in a distributed fashion, SE mode fits very well. For software that runs on Linux, BSD or Android, FS emulation mode is likely a better fit. The downside of that is that creating a framework such as the one described in Section 4.1, one has to do multiple runs with the same settings to minimise overhead factors as also described in that section. For software that falls outside of the named constraints, such as software running on Windows Embedded or require FS mode but with a different operating system than listed above, *gem5* simply cannot be used as these modes of operation are not currently supported.

- RQ3: Does *gem5* currently provide the facilities to do a full WCET analysis for *PapaBench*?

During this research, a number of issues were identified with running the *PapaBench* benchmark:

- The Paparazzi project, upon which *PapaBench* is based on, includes hardware simulation using AADL source code. This means that the autopilot and fly_by_wire executables get mapped into using the same RAM areas and communicate with each other through that. *PapaBench* however, creates two separate Linux executables that do not use and shared memory mechanism. The communication between the two executables therefore is not available.
- Much like the previous point, interrupt handler functions are present in the C source code, but they are never being called from anywhere. The AADL hardware specification connected the functions with proper interrupts.
- An example task where these two points become a problem for execution time analysis is task 3, *send_data_to_autopilot_task*. This task is intended to send out some last radio signal values over SPI from the fly_by_wire process to the autopilot process. Sending bytes over SPI normally triggers interrupts that the sending processor has to handle, which can happen during the execution of the current task. A similar process happens on the receiving processor, which has to store incoming data in a buffer, to be read at a later time.

The intended behaviour of *PapaBench* is one we have not been able to simulate as *gem5* does not currently, to our knowledge, offer an architecture which includes SPI or UART devices nor offers connecting user space functions with interrupts of those devices and the processors. This necessarily means that any execution time distribution created with *gem5* is not the worst case execution for *PapaBench*. However, we do have to remark that the suitability of *PapaBench* to our scenario exceeded our expectation. Were it not for the breadth of functions emulated in *PapaBench*, we would likely not have found some of the aforementioned issues.

6.3 Threats to validity

- Although care has been taken to increase accuracy of the simulations, there are a lot of options that have not been assessed in this paper. Examples include the lack of taking shared cache into account, proper task dependency management e.g. starting tasks only when a previous task has finished and variation of when a task starts relative to another task on a separate core as well as task synchronisation methods, to ensure tasks are scheduled when other tasks are running on the system.

- *PapaBench* aims to be a real-world benchmark, however the benchmark mode does not include proper cross-task communication. Moreover, the interrupts in the program are available in the source code, but never called. For this paper, we accepted this state because the focus is on creating a baseline and that this point would be a good fit for future research. Furthermore, the edits made to the benchmark as well as our given setup make it hard to compare the results to external runs of *PapaBench*.
- Internal validity is also somewhat lacking in our framework. While we did what we call a validation run, this validation run is also just a subset of the entire state-space of *PapaBench*, reduced task set or otherwise. Reducing the task set further would allow us to analyse the entire state-space, but reduce the diversity of tasks which would then be a threat to validity.
- Moreover, the results of this paper are certainly tailored to a niche: software tasks that run on a POSIX system, a general lack or at least a very small number of system calls and no disk I/O in the tasks. For embedded software this works, but outside that, it is likely that our results fail to hold. On top of that, our research is specifically for the dual-core *PapaBench*, further narrowing our niche.

6.4 Further research

As mentioned in Chapter 3, more time should be spent on validating our educated guess that background processes and the CPU scheduler are the main reasons which increase the simulation time of the benchmarks in FS mode. Running the benchmarks under real-time settings under Linux would be one approach we would suggest, another being synchronising task start on all cores.

A better approach to measuring execution times in multi-core settings with embedded software might actually not be with *gem5*. Given the aforementioned constraints on interrupts, differing hardware architectures and non-Linux simulations with *gem5*, it might be worthwhile to check if a similar setup as ours can be used with a tool like AADS [14]. Having control over the definition of the hardware as well as the software would likely yield a more accurate execution time distribution than can be achieved with *gem5*.

Another interesting point is that while we test task order influencing on multiple cores in our setup, which should be noticeable due to only using 1 memory rank in *gem5*, it seems that this is not that great of an impact. Modifying variables the influence execution time seems to impact execution times noticeably more than when tasks are scheduled relative to another task, on a different core. This is likely the result of only using L1 caches: each core only requests intermittent batch access to instructions and data for reading. Perhaps when dealing with a program that writes a lot more to memory or when introducing L2 caches, which we only used in our SE vs FS comparison due to FS mode requiring it, would yield more explanations as to why this appears to be so.

Bibliography

[1] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[2] Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean-Paul Bahsoun, and Marianne De Michiel. PapaBench: a Free Real-Time Benchmark. In Frank Mueller, editor, *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*, volume 4 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2006. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[3] B. Braams, S. Altmeyer, and A. D. Pimentel. Edify: An execution time distribution finder. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2017.

[4] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In Martin Schoeberl, editor, *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, volume 55 of *OpenAccess Series in Informatics (OASIcs)*, pages 2:1–2:10, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[5] EEMBC. EEMBC Autobench. http://www.eembc.org/benchmark/automotive_sl.php. [Online; accessed 24-July-2018].

[6] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem&mdash;overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.

[7] A. Butko, R. Garibotti, L. Ost, and G. Sassatelli. Accuracy evaluation of gem5 simulator system. In *7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, pages 1–7, July 2012.

[8] F. A. Endo, D. Couroussé, and H. Charles. Micro-architectural simulation of in-order and out-of-order arm microprocessors with gem5. In *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, pages 266–273, July 2014.

[9] Ashkan Tousi and Chuan Zhu. Arm research starter kit: System modeling using gem5, 2017.

[10] gem5. Supported Architectures - gem5. http://gem5.org/Supported_Architectures. [Online; accessed 23-July-2019].

[11] Vrije Universiteit Amsterdam et al. DAS-4: Distributed ASCI Supercomputer 4. https://www.cs.vu.nl/das4/. [Online; accessed 10-July-2019].

[12] H. Bal, D. Epema, C. de Laat, R. van Nieuwpoort, J. Romein, F. Seinstra, C. Snoek, and H. Wijshoff. A medium-scale distributed system for computer science research: Infrastructure for the long term. *Computer*, 49(5):54–63, May 2016.

[13] Skibiński et al. Yann Collet, Przemysław Skibiński. Zstandard - Real-time data compression algorithm. https://facebook.github.io/zstd/. [Online; accessed 23-July-2019].

[14] R. Varona-Gómez and E. Villar. Aadl simulation and performance analysis in systemc. In *2009 14th IEEE International Conference on Engineering of Complex Computer Systems*, pages 323–328, June 2009.