Slow and Steady: Measuring and Tuning Multicore Interference

Dan Iorga
Imperial College London, UK
d..iorga17@imperial.ac.uk

Tyler Sorensen
UC Santa Cruz, USA
Princeton University, USA
tyler.sorensen@ucsc.edu

John Wickerson
Imperial College London, UK
j.wickerson@imperial.ac.uk

Alastair F. Donaldson
Imperial College London, UK
afd@imperial.ac.uk

Abstract—**Now ubiquitous, multicore processors provide replicated compute cores that allow independent programs to run in parallel. However, shared resources, such as last-level caches, can cause otherwise-independent programs to interfere with one another, leading to significant and unpredictable effects on their execution time. Indeed, prior work has shown that specially crafted *enemy programs* can cause software systems of interest to experience orders-of-magnitude slowdowns when both are run in parallel on a multicore processor. This undermines the suitability of these processors for tasks that have real-time constraints.**

In this work, we explore the design and evaluation of techniques for empirically testing interference using enemy programs, with an eye towards reliability (how reproducible the interference results are) and portability (how interference testing can be effective across chips). We first show that different methods of measurement yield significantly different magnitudes of, and variation in, observed interference effects when applied to an enemy process that was shown to be particularly effective in prior work. We propose a method of measurement based on percentiles and confidence intervals, and show that it provides both competitive and reproducible observations. The reliability of our measurements allows us to explore *auto-tuning*, where enemy programs are further specialised per architecture. We evaluate three different tuning approaches (random search, simulated annealing, and Bayesian optimisation) on five different multicore chips, spanning x86 and ARM architectures. To show that our tuned enemy programs generalise to applications, we evaluate the slowdowns caused by our approach on the AutoBench and CoreMark benchmark suites. Our method achieves a statistically larger slowdown compared to prior work in 35 out of 105 benchmark/chip combinations, with a maximum difference of 3.8×. We envision that empirical approaches, such as ours, will be valuable for 'first pass' evaluations when investigating which multicore processors are suitable for real-time tasks.

I. Introduction

Multicore processors have become thoroughly mainstream, and are now routinely used even in safety-critical settings [31]. Yet, in a real-time domain, the greatest asset and defining feature of multicore architectures – that they allow work to be distributed among multiple processing cores working simultaneously – begets a significant weakness: interference between cores.

This interference arises from contention between cores for shared resources (e.g. caches, buses, and main memory), and it can have a significant and unpredictable effect on the worst-case execution time (WCET) of programs [24]. This undermines the suitability of multicore processors in scenarios that involve real-time constraints.

This paper is concerned with *empirical* techniques for estimating upper bounds on how much interference a program running on a multicore processor could be subjected to. Although empirical techniques cannot promise to identify the absolute worst case, they are nonetheless valuable for making rapid comparisons between candidate processors. The key idea is to design *enemy programs* that access shared resources in just the right way to maximise the contention with a program running on another core [24], [21], [12]. Prior work has designed several enemy programs, demonstrating substantial slowdowns on commercial off-the-shelf (COTS) processors of more than 300× [3].

The first contribution of this paper is a method for reliably estimating how much interference these enemy programs really cause. We propose a two-pronged measurement strategy that directly mitigates against factors that can be controlled (e.g. frequency throttling and thread migration) and uses statistical methods to mitigate against factors that are more difficult to control. We apply our measurement strategy to enemy programs proposed in prior work, and show that while there is no doubt that very large slowdowns are possible, it is difficult to reproduce the same magnitudes of slowdown once the various potentially-confounding factors are taken into account.

Thus, armed with a method for reliably measuring multicore interference, our second contribution is to investigate the extent to which *auto-tuning* techniques can be effective at generating better enemy programs. To this end, we have designed and implemented an open source tool that automatically tunes the parameters of a set of ANSI-C enemy programs, aiming to maximise the interference they cause on shared resources. Our tool supports three different auto-tuning strategies – random search, simulated annealing and Bayesian optimisation – and we compare the effectiveness of each. We have applied our tool to create enemy programs for five different chips, listed in Table I.

We evaluate our approach using benchmarks from the CoreMark [6] and AutoBench [2] real-time application suites. Our auto-tuning framework is able to create enemies that cause slowdowns up to 3.8× larger than the slowdowns caused by enemies studied in prior work that were hand-tuned by experts. More generally, our results show that auto-tuning for about 10 hours can lead to statistically significant improvements in

TABLE I: Development boards used to evaluate our approach

Name	Short name	SoC	Arch	Cores
Raspberry Pi 3 B	Pi3	BCM2837	Arm A53	4
DragonBoard 410c	410c	Adreno306	Arm A53	4
Intel Joule	570X	570x	Atom	4
Nano-PC T3	T3	S5P6818	Arm A53	8
BananaPi M3	M3	A837	A7	8

observed slowdowns for 35 out of the 105 benchmark/board combinations that we tried. We attribute the effectiveness of auto-tuning here to its ability to uncover parameters that exploit architectural features that are not immediately obvious and are thus unlikely to be discovered even by experts. A system developer can use these enemies with instrumentation and measurement tools (e.g. performance counters), to gain more insights into the system. An added benefit of our methodology over expert-tuned enemies is that it is *portable*: it can be used to automatically tune effective enemies across multiple, distinct architectures. We emphasise that auto-tuning is only enabled once our reliable measurement methodology is in place. Without this, the slowdowns associated with different enemy program configurations cannot be meaningfully compared.

In summary, our main contributions are:

- a set of principles for how enemy programs can be rigorously evaluated, with emphasis on reproducibility (Section II),
- an auto-tuning framework that can automatically configure enemy programs to maximise interference without the need for expert knowledge (Section III), and
- an experimental study instantiating our auto-tuning framework with our reliable measurement method and using it to tune enemy programs that have been proposed in prior work, obtaining enemies that can be up to 3.8× more effective at provoking interference in real-time applications (Section IV).

Our tool is open source and available online.[1]

II. Measuring Interference Reliably

We first present our approach for measuring the interference caused by an enemy program in a reliable manner, which underpins our practical instantiation in Section IV of the auto-tuning approach that we present in Section III.

As in prior work [24], [3], we are interested in evaluating *commodity* hardware using a *regular* (i.e. non-real-time) operating system, as this is a realistic usage scenario for early evaluation of COTS processors. But *unlike* this prior work, which pays little-to-no explicit attention to measurement reliability (i.e. variance and uncontrolled interference), we go to great lengths to stabilise the system environment. This significantly lowers the amount of measurement variance, which in turn allows us to produce results that have high statistical confidence. To this end, we rely on two principles: (1) controlling as many sources of interference as possible, and (2) acknowledging that not everything can be controlled and using statistics to account for any remaining interference.

Section II-A describes our unsuccessful attempt to reliably replicate prior work in an uncontrolled environment. Sections II-B and II-C then show how these results can be made more reproducible by following our first principle, and our first and second principles, respectively. The net effect of applying our approach is to "tame" the slowdowns achieved by the enemy programs of prior work so that they yield smaller, but still impressive, slowdowns that can be reliably reproduced, and that are thus more likely to be genuinely attributable to the interference effects of enemy programs.

We attempted to reproduce the highest slowdowns reported in recent interference work by Bechtel and Yun [3]: namely, that their BwWrite enemy program can cause a slowdown of more than 300× on a synthetic memory-intensive piece of software (on a Raspberry Pi 3 B chip). The architectural intuition behind these results is that the BwWrite program exploits the internal structure of non-blocking shared caches by filling the miss-status holding register and write-back buffer.

The section is structured around a discussion of Figure 1, which has six bars, (a) through (f). Bar (a) shows the results for the BwWrite enemy program reported by Bechtel and Yun in [3]. Bars (b) through (f) show results for our efforts to reproduce their results under successively more controlled conditions, with (b) using an uncontrolled environment at one extreme, and (f) a controlled environment combined with statistical techniques.

A. Uncontrolled environment

The experiment described in [3] was performed on the Pi3 developer board, with one core executing the synthetic memory-intensive software and the other cores executing instances of the BwWrite enemy program. For simplicity, we shall henceforth refer to a setting where a program under test is running on one core and the other cores are executing instances of the BwWrite enemy program as the Bechtel and Yun Environment (BYE).

The exact operating system used in [3] is not specified, so we installed a clean version of *Raspbian Buster Lite* on the Pi3– the minimal install available for this development board. Since Bechtel and Yun [3] do not discuss efforts to control the environment, we also did nothing in this regard at this stage.

We *are* interested in configurations that produce high interference only under specific timing conditions, and thus might produce such high interference only rarely. Because of this, we need to measure the same configuration multiple times. Bar (a) of Figure 1 shows the slowdown reported in [3], while bar (b) shows the results we obtained trying to replicate this result in our freshly installed (uncontrolled) environment. We ran each experiment 20 times, taking 25 measurements each time and recording the maximum value. The error bars represent the variation of this maximum across the 20 runs. We were able

[1] https://github.com/mc-imperial/multicore-test-harness

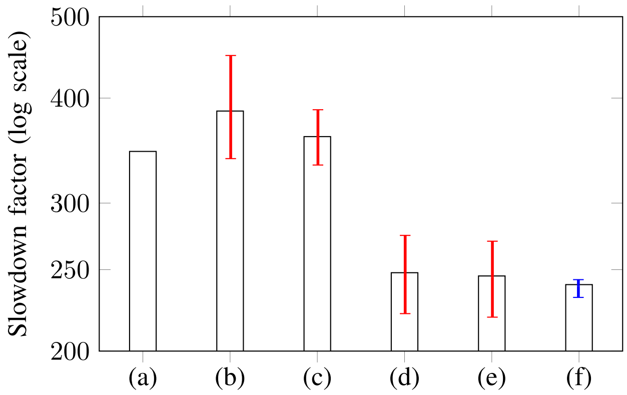


Fig. 1: Slowdowns obtained using the BwWrite enemy program on the Pi3: (a) as reported by Bechtel and Yun [3], and then reproduced by ourselves (b) in an uncontrolled environment, (c) in an environment where just frequency throttling is accounted for, (d) in an environment with controlled frequency throttling and frequency scaling, (e) in a fully controlled environment, and (f) using our statistical approach. The graph uses a logarithmic scale. Error bars for (b) to (e) show the range of maximum values observed. Error bars for (f) show the confidence interval around the percentile.

to obtain results that are similar to those reported in [3], but with *very* high variance.

Under the hypothesis that the environment in which experiments are performed can significantly affect performance measurements, we proceed to incrementally clean up (i.e. reduce interference) from the environment.

B. Controlled environment

Taking inspiration from previous work that measures microarchitectural details through microbenchmarking [10] (but not in the context of real-time systems), we first consider mitigation of factors related to the hardware and operating system that might cause us to wrongfully conclude that observed slowdowns are due to the effects of enemy programs.

1) Hardware: Hardware mechanisms are generally designed to be transparent to the user and can be unpredictable. We take into account two factors here: frequency throttling and cache warmth.

a) Frequency throttling: When the temperature of a processor increases beyond a certain limit, the processor can throttle its clock frequency to reduce the temperature. As an example, Figure 2 shows how frequency is affected by temperature on the Pi3. The data was gathered for a run of a cache-intensive workload (though in our experience, similar results can be obtained using any computationally-intensive workload) using the *vcgencmd* tool. The figure demonstrates that frequency throttling can cause serious fluctuations in performance measurements. To mitigate the risk of mistaking throttling effects for enemy-induced interference, we measure temperature reported by the operating system at the end of each experiment and discard the result if the reported temperature is above 80 degrees Celsius. We empirically found that using this temperature threshold works well on all the processors used in our experiments.

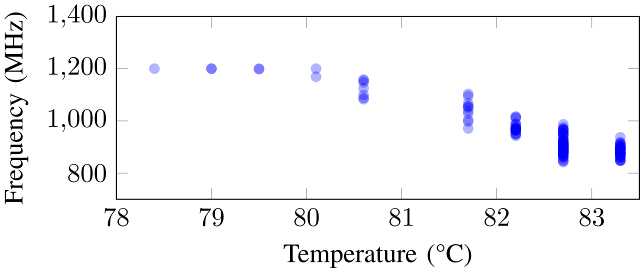


Fig. 2: Frequency variation due to temperature on the Pi3. As the execution temperature increases, frequency throttling is activated, slowing down the core and making it difficult to determine the cause of the slowdown in execution time.

Bar (c) of Figure 1 shows results for the BwWrite example when we control for frequency throttling only. This decreases variation by 56%, but has a small and not statistically significant effect on the maximum slowdown factor (the confidence intervals associated with bars (b) and (c) overlap).

b) Cache warmth: We flush the cache at the beginning of each experiment to guard against the possibility that data left over from one experiment might affect the execution time of the next experiment.

2) Operating System: Another source of unreliability in execution time is the preemption mechanisms of modern operating systems. A preempted victim may have a longer execution time due to the interrupted time, as well as potential cache pollution by the interrupting task. This interference should not be attributed to the enemy processes.

To maximally mitigate against measurement error due to preemption we would have to run experiments in a "bare-metal" fashion or via a proven real-time operating system. However, this would destroy the portability of our approach, and the manual effort required per processor would defeat the main intended use of our work as a early-stage method to help in the selection of COTS processors for real-time systems. We thus run experiments under Linux, taking account of: thread migration, preemption, scaling governance, and ensuring parallel execution.

a) Scaling governance: We ensure that the operating system does not adjust the processor clock frequency by using the "powersave" non-dynamic governor.

Bar (d) of Figure 1 shows results associated with the BwWrite enemy program when the governor is set to "powersave". Comparing bars (c) and (d) we see that this more steady frequency management setting decreases the maximum detected slowdown by 33%.

b) Thread migration: To guard against the operating system migrating threads between cores, we pin the program under test (PUT) and enemy programs to specific cores using the `taskset` Linux command.

c) PUT preemption: We run the PUT at the highest available priority level to reduce the risk that our measure-

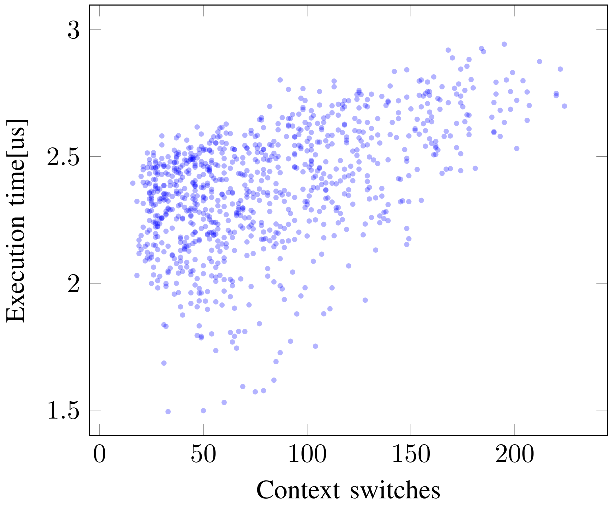


Fig. 3: The effect of context switches on the execution time on the Pi3

ments are affected by the cost of context switching. We now discuss an empirical investigation into the impact of, and our approach to reduce the number of, context switches. We use the `getrusage` command, which returns the number of context switches a process had during execution.

First, we determine if context switches have a statistically significant impact on execution time. To do this, we measured the context switches for the experiment in Figure 1. We plotted the samples using the number of context switches as the x-axis and the execution time as the y-axis. Visually we can see a clear correlation, and this can be statistically confirmed using the Spearman's rank-order correlation test which shows a strong positive correlation.

Next, we investigate whether setting the scheduling of the PUT has a statistically significant effect on reducing the number of context switches. We ran the same experiment as the previous paragraph, however, this time the priority of the PUT is set to the maximum allowed value. To determine if this scheduling scheme meaningfully affects the number of context switches, we compare the results using the Wilcoxon rank-sum method [20], a non-parametric test to determine if one set is stochastically larger than another set. The test confirms that the number of context switches is significantly reduced, returning a p-value of less than .05. Concretely, the median number of context switches observed per PUT execution without priority scheduling is 88, and it decreases to 75 when the maximum priority is set.

At the end of this subsection, we will return to Figure 1 and show that while context switching (and other environmental factors, discussed next) do correlate with execution time, their overall impact on slowdowns (and variance) is small compared to the previously discussed frequency throttling and power governance.

d) Ensuring parallel execution: Our measurements will be meaningless if the enemies do not actually run in parallel with the PUT, so we need to take care with respect to the process startup latency. To evaluate the maximum startup latency on each platform, we used a latency evaluation framework [32] and discovered that the maximum startup latency is generally below 1 ms. Thus, we conservatively allow 10ms between launching enemy programs and launching the PUT.

Bar (e) in Figure 1 shows the slowdown obtained when the environment is controlled as described above. We can see that smaller slowdowns are obtained, but that there is less variation between measurements. The values reported in prior work [3] are between the ones obtained in the uncontrolled environment and the controlled environment. This highlights how difficult it is to reproduce an experiment without knowing the exact configuration in which it was run.

To summarise the findings in this section, frequency throttling and scaling are the key factors that influence the degree of slowdowns and the variation. While the other environmental factors, such as scheduling priority, do have a correlation to the PUT execution time, they are not statistically significant when compared to frequency throttling and scaling, as can be seen in Figure 1, bars (b)-(f).

We have presented an accessible approach, e.g. that can be straightforwardly applied to COTS systems, for controlling the execution environment. However, as we note, we cannot account for all possible sources of interference, e.g. context switches, and they do correlate with execution time, e.g. see Figure 3. Our efforts to control such factors are impactful (see Figure 1), however, we still observe a high variance. Thus, we next present a statistical approach that aims to decrease the variance, i.e. increase measurement precision, even in the presence of uncontrollable environment factors inherent in COTS systems.

C. Statistical approach

If the remaining variance in experimental measurement illustrated by the bar (e) of Figure 1 was *entirely* due to the interference induced by the enemy programs, then taking the *maximum* of the interference measurements would be appropriate, as prior works have done.

However, despite the stringent efforts described in Section II-B, it is not possible to fully eliminate nondeterministic aspects of the environment that might interfere with our measurements. For example, without extreme manual effort it is not practical to ensure that system daemons have been completely disabled.

We thus need a statistical analysis over multiple measurements that does a reasonable job of estimating the maximum interference caused by enemy programs, ignoring outliers arising due to external system events that we have not been able to control.

Figure 4 illustrates how outliers can affect the measurement of interference caused by enemy programs. The figure shows the execution speed of a vector addition program executing on the 410c (see Table I), running alongside various random

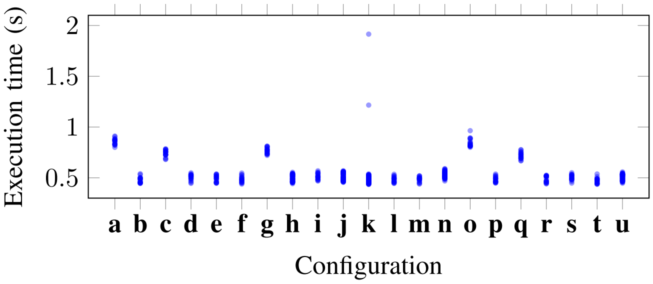


Fig. 4: Each point shows the execution time of a simple vector addition running alongside various enemies on the 410c.

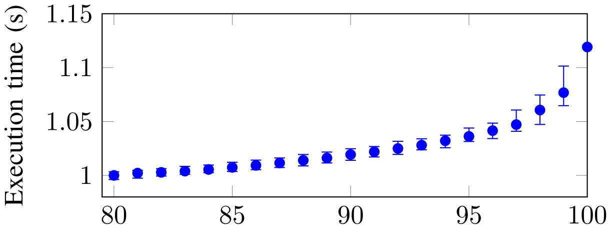


Fig. 5: The 95% confidence interval for each percentile after 1000 measurements on the Pi3, when causing interference to a vector addition program.

enemy program configurations.[2]

We ran the vector addition program 20 times in the presence of each enemy program configuration **a** through **u**. The 20 points above each configuration in Figure 4 show, for each run, the execution time that was observed. The two unusually high execution times associated with enemy configuration **k** are clearly suspicious. While it is *possible* that these high times really are due to rare, timing-related interference between enemy configuration **k** and the program under test, it is prudent to assume that these outliers are blips caused by uncontrolled external factors. If we would measure interference as the *maximum* slowdown observed over all runs, we would certainly select configuration **k** as the most effective of the enemy program configurations in Figure 4, discarding the more readily-reproducible results associated with e.g. configuration **o**.

To account for such outliers, we aim to measure interference with respect to a percentile that is as close to the 100^{th} percentile as possible, but such that the effects of rogue outliers are eliminated. The q^{th} percentile of a data set is defined as the value where $q\%$ of the data is below that value. We take inspiration here from prior work on comparing latency and end-to-end times of Linux schedulers [7].

To choose a reasonable value for the percentile, we ran a pilot experiment consisting of the previously mentioned vector addition application alongside three enemy programs. We measured the execution time 1000 times and calculate the 95% confidence interval for the 75^{th} to 100^{th} range of percentiles. The confidence interval gives an estimated range of values which is likely to include the true value, the estimated range being calculated from a given set of sample data. The results of this experiment are highlighted in Figure 5 where we can observe that as we increase the percentile, we also have to tolerate a wider confidence interval. At this point, we have to make a trade-off between wanting to be confident in the accuracy of our measurements and wanting not to ignore interesting cases. We ran the same experiment on the other boards at our disposal and have chosen the 90^{th} percentile as a reasonable trade-off.

Returning to the BYE, bar (f) in Figure 1 shows the results obtained after introducing our statistical approach. Comparing bars (e) and (f), the slowdown reported when outliers are removed is *slightly* more modest, but the confidence interval is *substantially* smaller. Such small confidence intervals allow us to reliably compare multiple enemy processes in terms of interference caused on a program under test.

Summary: To recap again what Figure 1 shows with respect to the slowdowns reported by Bechtel and Yun [3] for their BwWrite enemy program: bars (a) and (b) indicate that we *can sometimes* reproduce the same order-of-magnitude slowdown, but with extremely high variance; bars (c), (d) and (e) show how we progressively improved the control of our environment; finally, bar (f) shows that, by using statistical methods to prune outliers, these more modest, yet still impressive, slowdowns can be observed with much lower variance.

Armed with this rigorous method for measuring slowdowns, we now propose an automated technique for tuning enemy programs to maximise interference.

III. Tuning Hostile Environments

We now describe our method for tuning a *hostile environment*—a set of enemy programs, each of which will run on its own distinct core in parallel with a PUT—with the aim of developing a hostile environment that interferes maximally with the PUT. In this section we describe our approach in general terms, abstracting away specifics such as the shared resource with which interference is associated and the search strategy used to guide the tuning process. We provide a concrete instance of this framework, with detailed experimental results, in Section IV.

Section III-A details the two-stage nature of our tuning approach. In the first stage, we use auto-tuning to search for enemy template parameters per resource. In the second stage, a more generic hostile environment is constructed from the previously-tuned enemy programs. In Section III-B we discuss the tractability of single- and multi-stage tuning strategies and various assumptions we have made.

A. Our Two-Stage Tuning Methodology

A user of our approach must first decide on the interference paths they wish to investigate, i.e. shared resources for which multicore contention might cause slowdowns. For each one

[2] These configurations were produced by the random search auto-tuning strategy that we discuss in Section IV-B, but for the purposes of the present discussion all that matters is that each represents a distinct configuration of enemy programs whose interference effects we are interested in measuring.

of these paths they must create: (1) a parameterised *enemy template* that will run in an infinite loop and stress the resource and (2) a *victim program* that performs a fixed amount of synthetic work, making heavy use of the resource. This is the main manual task associated with using the method we present, and the results obtained by our framework will only be meaningful if the user provides suitable enemy and victim programs. We detail example enemy and victim programs that we have found to work well for a number of shared resources in Section IV and others that did not work equally well in Appendix A.

Our tuning methodology has two stages. In Stage 1 we automatically tune the parameters of each enemy template with the objective of maximising the effect of the enemy on its corresponding victim program. Our framework is parametric with respect to both the search strategy used and the specific notion of "effect" that determines the objective function. We discuss practical choices of search strategy in Section IV-B, and use the statistically-rigorous measurement approach of Section II-C as the objective function in practice. Because the victim program is vulnerable to interference on the target resource, the degree to which it is affected serves as a proxy for measuring interference on the associated interference path.

Alternatively, the real PUT can be used as a tuning victim. However, this would only be practical if the PUT is (a) known in advance, and (b) sufficiently fast-running to be executed a large number of times during the tuning phase. In Section IV-D we present results for experiments that explore the trade-offs associated with PUT tuning.

Stage 2 combines the resource-specific tuned enemy programs found in Stage 1 to construct a hostile environment that aims to be effective across all considered resources. That is, because we aim to interfere with an arbitrary PUT, which may utilise multiple shared resources in complex ways, we search for a combination of tuned enemy programs that is effective across *all* victim programs. Rather than tune enemy program parameters across multiple victim programs (we argue in Section III-B that this is not tractable for large search spaces), we instead exhaustively search for a Pareto-optimal combination of individually-tuned enemy programs.

1) Stage 1: Tuning Enemy Programs per Resource: We now rigorously describe how, for a given chip, the parameters for an enemy template are tuned to maximise interference with the corresponding victim program.

In what follows, let R denote the set of resources to be targeted. For every $r \in R$, a victim program (V_r) and enemy template (T_r) must be provided. We note that while R can contain arbitrary sets of resources, it is up to the user to design these judiciously. When tuning for memory hierarchy interference as in this paper, we use $R = \{\mathsf{cache}, \mathsf{memory}\}$.

A template T_r takes a set of parameters drawn from a parameter set P_r. For a given parameter valuation $p \in P_r$, let $T_r(p)$ denote the concrete enemy program obtained by instantiating T_r with p.

Let us define an *environment* as a set of programs, each of which can be assigned to a core. If these programs are enemy programs, we say that the environment is *hostile*. For a victim program V_r and environment e, let $\mathsf{perf}(V_r, e)$ denote the performance associated with executing V_r on core 0, in parallel with e running across the remaining cores. For the sake of generality, we leave the specific definition of perf abstract here. In practice, it could measure wall-clock execution time, readings obtained from performance counters, or consumption of some other resource, e.g. energy.

For a template T_r and parameter valuation $p \in P_r$, let $\mathsf{itf}(V_r, T_r(p))$ denote a numeric metric value for the *interference* associated with (1) executing V_r in isolation on core 0 (with all other cores unoccupied), compared with (2) executing V_r on core 0, in parallel with an instance of $T_r(p)$ on every other available core. That is:

$$\mathsf{itf}(V_r, T_r(p)) = \frac{\mathsf{perf}(V_r, T_r(p))}{\mathsf{perf}(V_r, \mathsf{nop})}$$

where nop denotes the absence of an enemy process. The aim in the first stage is to determine the parameters that maximise interference to the victim program as follows:

$$\underset{p \in P_r}{\arg\max}\ \mathsf{itf}(V_r, T_r(p)) \tag{1}$$

Because P_r is too large to search exhaustively, search strategies can be used to approximate the maximum. In Section IV-B we evaluate several natural choices: random search, simulated annealing, and Bayesian optimisation.

Let p_r^{tuned} denote the best parameter setting that was found via search using the chosen strategy. We refer to the set $\{T_r(p_r^{\mathsf{tuned}}) \mid r \in R\}$ as the set of *resource-tuned enemies*.

2) Stage 2: Tuning a Generic Hostile Environment: Stage 1 considers the *same* enemy running on every available core other than that occupied by the PUT. We now aim to devise a deployment of enemy programs that is effective at inducing interference across all resource types since we do not know the resource usage profile of the PUT a priori. We determine the best configuration of enemy programs by using a strategy similar to the one described by Sorensen et al. [27]. In particular, for an n-core processor where the PUT executes on core 0, we seek a hostile environment that maps each core $i \in \{1, \ldots, n-1\}$ to a resource-tuned enemy.[3]

We now describe our strategy for choosing a suitable hostile environment from the set of $|R|^{n-1}$ possibilities. First, for each resource $r \in R$, we rank all possible hostile environments according to the extent to which they cause interference with V_r, i.e. according to $\mathsf{itf}(V_r, e)$, with the environment that induces the largest interference ranked first. Let $\mathsf{RankedEnvironments}(r)$ denote this ranking.

While the number of possible hostile environments grows exponentially with the number of cores, for three resources it is feasible to exhaustively search the space of 8 and 128 candidate environments associated with 4- and 8-core chips,

[3] We also considered leaving cores unoccupied to account for the possibility that multiple enemies might interfere with each other, in which case interference might not always be maximised by filling up all cores. However, this does not seem to be the case in practice.

respectively. We discuss the use of approximations to scale to larger core counts in Section III-B.

We then select a Pareto-optimal hostile environment. This is an environment e for which there does not exist an environment $e' \neq e$ such that for all $r \in R$, e' is ranked more highly than e in $\mathsf{RankedEnvironments}(r)$. Being Pareto-optimal, e may not be unique. In such cases, we break ties by selecting the environment that is ranked better in all but one of the $\mathsf{RankedEnvironments}(r)$. In our experimental results, there were no ties after this step.

B. Assumptions and Tractability

We now describe several assumptions the above tuning methodology makes about multicore interference and comment on some aspects of the tractability of the approach.

1) Multi- vs. Single-Stage Tuning: We have described a two-stage process, in which enemies are first tuned to cause slowdowns in their corresponding victims, and second, a Pareto-optimal combination is found across all victim programs. It is possible to formulate these stages into a single *bilevel optimisation* problem where the combination of enemy programs is the outer optimisation task, and individual enemy template parameters are the inner optimisation task.

While such a formulation is mathematically elegant, the parameter space quickly becomes very large. Additionally, as shown in Section IV-B, the search space in this domain is often unstructured, thus more intelligent searching techniques are not more effective than simple random sampling. Indeed, our pilot experiments using this approach failed to provide meaningful slowdowns after many hours of tuning.

The two-stage tuning approach allows more precise exploration into individual interference paths, which are interesting even by themselves, before being merged into a generally effective hostile environment.

2) Tuning With All Cores Occupied: In the first stage of tuning, we assign $n-1$ enemy programs, all with the same parameters, across all cores except one (which is used to run the victim program). However, in the second stage, we select a mapping where the enemy programs that were tuned on $n-1$ cores may execute on *fewer* cores. Thus, it is natural to wonder whether enemies tuned on $n-1$ cores remain effective when run on fewer cores.

Our decision to tune on $n-1$ cores is based on three factors. First, considering every possible core count for every resource during tuning would lead to a blow-up of the already very large parameter-tuning space associated with one core count. Second, hostile environments are noisy, making it challenging to measure interference reliably (as discussed in depth in Section II). As a result, we can reason more confidently about our measurements in cases where interference is most pronounced, i.e. when all available cores are occupied. Third, we have found that the following *monotonicity* property tends to hold in practice, where $r \in R$ is a resource, $p, p' \in P_r$ are parameter valuations, and $\mathsf{itf}(V_r, T_r(p), m)$ is the same as $\mathsf{itf}(V_r, T_r(p))$ except that $T_r(p)$ is executed on $m < n$ cores:

$$\begin{aligned}(\mathsf{itf}(V_r, T_r(p)) > \mathsf{itf}(V_r, T_r(p')) \wedge 0 < m < n)& \\ \implies \mathsf{itf}(V_r, T_r(p), m) > \mathsf{itf}(V_r, T_r(p'), m)&\end{aligned} \tag{2}$$

That is, if parameter valuation p beats p' when enemies are run on all available cores then p tends to *also* beat p' when enemies are run on fewer cores. We tested this property over a subset of possible configurations by sampling the parameter space of a cache enemy program (described in Section IV) for one hour on the Pi3, and found the statement to hold true for 93% of runs.

3) Scalability: Stage 2 exhaustively searches combinations of core-to-enemies mappings. This is tractable for the 4- and 8-core chips examined in this work, core counts that we understand to be typical for processors being considered for use in real-time systems. Stage 2 could be straightforwardly adapted to larger core counts by approximating a Pareto-optimal hostile environment using a search strategy, similar to the way search strategies are used to approximate the global maximum when tuning enemies per resource in Stage 1.

IV. An Instance of our Auto-Tuning Framework

We now describe how we have combined the auto-tuning methodology of Section III with the rigorous measurement approach of Section II to automate the process of creating a hostile environment that provokes interference on *shared memory* resources.

In Section IV-A we explain how we have generalized the enemy programs used in prior work by Bechtel and Yun [3] to create a parameterized enemy template suitable for auto-tuning, with an associated victim. We outline several natural choices for the search strategy that can be used to guide auto-tuning in Section IV-B. In Section IV-C we present experimental results related to creating a hostile environment with respect to synthetic victim programs, comparing the effectiveness of different search strategies for this purpose.

Having found a hostile environment that is particularly effective for each board, we evaluate the extent to which tuning with respect to synthetic victim programs translates to slowdowns on real-world benchmarks and compare to previous work in Section IV-D. Finally, in Section IV-E, we delve deeper into reasons for the effectiveness of our enemy programs by presenting data obtained via hardware performance monitor counters. Appendix A briefly details alternative enemy programs that we experimented with, that either proved to be less effective than our generalisation of the BYE or were targeting resources that we ultimately considered beyond the scope of this paper.

A. Creating enemy templates

For our instantiation, we focus on the cache and the main memory as the interference paths. Previous work has shown how different configurations can be used to affect these paths [3], [28], [24]. The size of the memory being accessed as well as the operations performed on that memory are all

TABLE II: The enemy/victim code, the tunable parameters and their corresponding ranges. HEADER is “;;” for enemy programs and “int i=0;i<LIMIT;i++” for victims. LLC denotes the size of last-level cache for the processor under test.

```
#define ACCESS_PATTERN(*scratch_addr) ...
volatile int8_t *scratch =
       (int8_t*) malloc(SCRATCH_SIZE);
for (HEADER)
  for (int i = 0; i += STRIDE;
              i < SCRATCH_SIZE)
    ACCESS_PATTERN(&(scratch[i]));
```

Parameter	cache **value**	memory **value**	**Enemy range**
SCRATCH_SIZE	LLC	10×LLC	1–5120KB
STRIDE	cache line size	cache line size	1–20
ACCESS_PATTERN	read,write	read,write	1–5 read/writes

parameters that impact the effect of the enemy process. Therefore we create a single enemy template by parameterising a concrete enemy program from previous work [3], and use this template to tune two distinct enemy programs: a cache enemy, tuned with respect to a cache victim, and a memory enemy, tuned with respect to a memory victim.

Table II summarises the code structure of the enemies and victims, the parameters used by the cache and memory victims (which are fixed for a given processor under test), and the range of parameters available for auto-tuning of enemies.

The code for victims and enemies is very similar, with the key difference being that an enemy program runs indefinitely, while a victim program runs for a fixed number of loop iterations. To this end, the HEADER parameter is instantiated in an infinite loop for the enemy and a finite loop for the victim.

The cache and memory victims iterate through the scratch buffer, using a stride equal to the cache line size of the processor under test (so that a large and diverse portion of the cache is accessed), performing a read from and a write to each buffer element under consideration. The victims differ only in the fixed choice of the SCRATCH_SIZE parameter. For the cache victim, it is set to the size of the last-level cache, denoted LLC in Table II. For the memory victim, it is set to 10 times the size of the last-level cache, denoted 10×LLC in Table II. Intuitively, the cache victim's working set should remain in last-level cache when there is no interference, so that this victim is prone to a slowdown when an enemy program causes last-level cache evictions. In contrast, the memory victim's working set does not fit in last-level cache, so that this victim will frequently access main memory and thus be prone to a slowdown when an enemy program causes interference on main memory. Our intent is that the cache and memory victims serve as proxies for PUTs that make heavy use of the shared resources of the LLC cache and main memory, respectively.

In contrast to the victims, the parameters SCRATCH_SIZE, STRIDE and ACCESS_PATTERN are available for tuning in the enemy template. This allows both last-level cache and main memory to be accessed in a chaotic manner that might interfere with the cache or memory accesses of the victim.

The framework is readily extensible with further victim and enemy templates to stress additional shared resources with associated tunable parameters.[4]

B. Search Strategies

Recall that the first stage of our tuning methodology (Section III-A1) requires a strategy to guide the search for effective enemy program parameters with respect to a victim. We intuitively expect the search space of enemy program configurations to be discontinuous with respect to interference, e.g. due to caches having fixed parameters that are typically powers of two, and memory being organised in banks. Therefore we use search strategies that do assume convexity of the cost function and do not rely on gradient information. We evaluate the following strategies:

Random search (RAN) This strategy repeatedly samples configurations and returns the best configuration that is observed. RAN is lightweight and provides a baseline against which to compare more sophisticated strategies. A weakness of RAN is that as it remembers nothing but the best combination of parameters seen so far, it may try the same configuration multiple times.

Simulated Annealing (SA) This strategy is a meta-heuristic to approximate global optimisation in a large search space [16]. SA is often used when the search space is discrete (e.g. all tours that visit a given set of cities), and can be a good match for problems where finding an approximate global optimum is more important than finding a precise local optimum in a fixed amount of time.

Bayesian Optimisation (BO) Having an unknown objective function, the Bayesian strategy is to treat it as a random function and place a prior over it [26]. The prior captures beliefs about the behaviour of the function. After gathering results of evaluating the function, which are treated as data, the prior is updated to form the posterior distribution over the objective function. The posterior distribution, in turn, is used to construct an acquisition function that determines what the next query point should be.

There are advantages and disadvantages to each strategy. RAN and SA can quickly determine the next query point. SA will concentrate its search near the best solution found towards the end of the search which might be advantageous if the search space does not have too many local maxima. RAN might get lucky and find a good solution, especially when

[4] We considered multiple shared resources however the cache and memory enemies seems to overwhelm all the others. We further detail this in appendix A.

TABLE III: Comparing search strategies. The strategies are ordered using the symbols described in Section IV-C. In each case, the best strategy is highlighted in bold, and the maximum slowdown obtained using that strategy is given in parentheses.

Board	Cache victim		Memory victim	
Pi3	SA<BO≲**RAN**	(51.5)	BO≲RAN≲**SA**	(16.1)
410c	SA≲BO≲**RAN**	(2.24)	RAN≲SA≲**BO**	(2.65)
570X	SA≲RAN≲**BO**	(2.07)	SA≲RAN≲**BO**	(2.5)
T3	SA<RAN≲**BO**	(6.6)	BO≲RAN≲**SA**	(5.7)
M3	SA≲BO≲**BO**	(14.1)	SA<RAN<**BO**	(21.52)

TABLE IV: The most aggressive hostile environments we found. A configuration such as 'VMCM' denotes that the four cores contain (respectively) the victim, a memory enemy, a cache enemy, and a memory enemy.

Board	Most aggressive hostile environment, per resource		
	Cache	Main Memory	Overall
Pi3	VCCC	VMMM	VMMM
410c	VMCM	VCMM	VCMM
570X	VMMC	VMMC	VMMC
T3	VMCM MCMM	VMCC CMCM	VMCM MCMM
M3	VCCC CMCC	VCMC MCMC	VCCC CMCC

the search space has little structure. BO needs time to remodel the objective function and the acquisition function after each new query is made. On the other hand, it is expected that BO will only sample points that will increase our knowledge of the problem. In general, one would expect to prefer the first strategies in cases where the cost function is cheap to evaluate and BO in cases where the cost function is expensive to evaluate. We evaluate the effectiveness of these approaches in Section IV-C.

Stopping Criteria: For all the tuning strategies, we need a reliable objective function. Our considerations described in Section II allow us to obtain a reliable value if multiple measurements are taken. For our instantiation, we measure 200 times with the same configuration and calculate the 95% confidence interval. The confidence interval is calculated in a distribution-free manner using the technique described by Hahn et al. [14]. If we have a minimum amount of measurements and the relative range of values within the interval gets small enough (we use 5%) before we finish the 200 measurements, we stop the measurements early.

C. Hostile Environment Construction

We now compare the search strategies described in Section IV-B and determine which one is best at finding effective parameters for our enemy template with respect to each of the cache and memory victims; this is the first stage of our tuning approach (see Section III-A1). We then turn to the second stage of our tuning approach (see Section III-A2) and construct hostile environments by combining enemies.

1) Effectiveness of Search Strategies for Tuning Enemies: We tune the enemy templates using their corresponding victim program as described in Section III-A1 with all three search strategies tuning for two hours. Since all search strategies have a certain degree of randomness and can sometimes get "lucky" (even BO starts by randomly sampling its starting points), we perform three runs of each search method for each shared resource. We use the Wilcoxon rank-sum method [20] to test whether values from one set are stochastically more likely to be greater than values from another set. This method is non-parametric, i.e. it does not assume any distribution of values, and returns a p-value indicating the confidence of the result.

The results of this experiment are shown in Table III, in which we construct an order of the effectiveness of each search method per resource, based on the best value found. Some orders are more confident than others: when the p-value is low (below or equal to 0.05) we have high confidence in the ordering, and we use the '$<$' symbol; when the p-value is high (above 0.05) we are not as confident in the ordering, and we use the '$\lesssim$' symbol. It seems that for most of the development boards, the tuning strategy does not impact the result that much and we cannot reliably say that one search strategy is better than the other. This is most likely due to the fact that the underling search space of the enemy templates is irregular and the smarter BO can not properly model it.

To search more thoroughly for enemy program parameters that maximise interference we tune each enemy template and its corresponding victim program with the winning search strategy for an extended period (10 hours). We empirically determined that this is long enough to reach the point of diminishing returns. Table III presents the maximum slowdown obtained alongside the search strategy used. An astute reader might notice that the slowdowns reported by prior work are substantially larger than our own. This discrepancy is due to a different metric being used. Prior work reported the time per access, while we reported the time for the entire synthetic application execution. Thus our reported slowdowns are amortised by the non-memory instructions. If we do the conversion from our metric to the time per access metric on the Pi3 for the cache resource, we observe a similar slowdown of 263×.

From Table I we see that the Pi3 and the 410c have the same architecture, but implemented in different SoCs. The Pi3 is especially vulnerable to cache interference while the 410c is much less prone to the same type of interference. It is likely that this can be explained by microarchitectural differences between the the two boards; however, we are not aware of the exact mechanism that causes this difference, since low-level details are generally not available for most commercial SoCs.

2) Constructing Hostile Environments: We now determine the Pareto-optimal hostile environment for each of the boards using the methodology described in Section III-A2. An example of this approach can be seen in Table IV, where we have listed the most aggressive environment for each development board and for each of the resources. In every case it happens to be either the top memory or cache enemy. Also, notice that combinations of cache and memory enemies are more effective (except for the Pi3) than just running the same enemy of each core. It can be non-intuitive to anticipate the effects of such combinations, motivating the need for such a technique.

D. Evaluating Hostile Environments on Benchmarks

The synthetic victim programs are designed to be especially vulnerable to shared resource interference. While these synthetic applications show extreme cases of interference, we are also interested in observing the effects on two industry-standard benchmark suites that are frequently used in the evaluation of embedded real-time systems:

EEMBC CoreMark is a standardised benchmark suite used for evaluating processors [6]. It runs the following algorithms back-to-back: list processing (find and sort), matrix manipulation (common matrix operations), state machine (determine if an input stream contains valid numbers), and CRC (cyclic redundancy check).

EEMBC AutoBench consists of automotive workloads, including: road speed calculation and finite impulse response filters [2]. This benchmark suite is of interest for the real-time industry and has been used in the evaluation of other works in this domain, e.g. [12], [11].

We want to see if our approach is statistically better than the original one that did not involve tuning. For this reason, we calculated the 95% confidence interval of the 90^{th} percentile for the benchmarks running in both our hostile environment and in the environment described by Bechtel and Yun [3]. We are again using our reliable metric presented in Section II. For simplicity, we shall henceforth refer our Tuned Hostile Environment as THE.

Figure 6 shows the result of these experiments. The centre of the ellipse is the 90^{th} percentile. The width is the BYE 95% confidence interval and the height is our THE 95% confidence interval. If the ellipses touch the diagonal, there is no statistical difference between the approaches. This is the case for the 410c and the 570X, where there seems to be no major improvement. However, there are many cases on the Pi3, T3 and M3 where our approach is able to cause more interference than the manual approach. This is without providing any extra architectural insight to the framework.

Our approach seems to be particularly effective on the T3 and the M3, the 8-core boards at our disposal. In these boards, the cores are groups into clusters of 4 that share a cache and there is no cache shared between all of the cores. For such a non-trivial architecture it is more difficult to hand-craft enemy processes that can fully exploit the shared resources

Overall, THE produced higher slowdowns in 35 application/board pairs (33%); BYE produced higher slowdowns in 6 application/board combinations (6%). The maximum difference observed between BYE and THE was 3.8× on the Pi3.

Tuning for an PUT: Our approach of tuning with respect to victim programs that serve as proxies for how an PUT might use a shared resource is appealing if a specific PUT of interest is not yet available, if there are many PUTs of interest, or if the PUT of interest cannot be executed rapidly. However, if a single, fixed PUT of interest is available and can be execute rapidly, our approach can be used to tune exactly for that PUT. This would make use of our rigorous measurement approach and our enemy process templates, but would not require our victim templates.

Because the benchmark programs considered in this work do execute relatively rapidly, we were able to run experiments tuning enemy programs specialised per benchmark program. Overall, and as expected, benchmark-tuned enemies are at least, and sometimes more, effective than enemies tuned with respect to victim templates. On the Pi3 the differences are quite noticeable, leading to slowdowns of up to 11×, whereas THE was able to cause slowdowns of up to 8×. However, on boards such as the 570X and the 410c, the differences were not that significant, with the slowdowns being within the confidence interval of the slowdowns caused by the more general THE.

E. Performance Monitor Counters

To get a better understanding of why our approach is effective, we access the performance monitor count (PMC) registers on Pi3 where our approach was more effective and, as such, expect the differences to be easier to analyse. We are especially interested in cache misses, bus traffic and main memory accesses. Unfortunately, there are no PMCs that can give us more insight into main memory access. However, there are PMCs for TLB accesses, which may shed light on main memory contention. We reran the experiment from Section IV-D but we also measured the performance counters using the Linux tool `perf`. We evaluated PMC values for our THE and the BYE one.

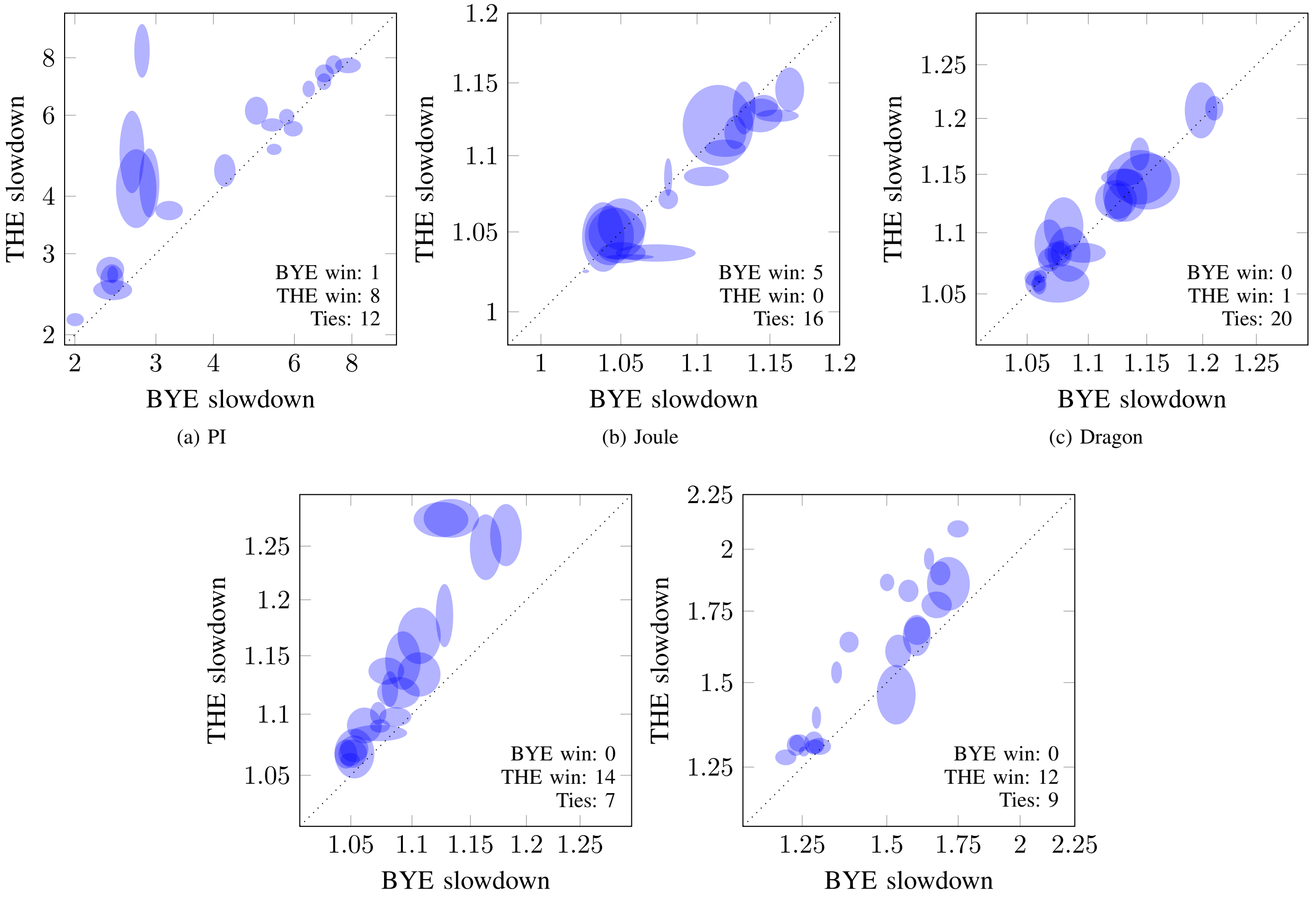
Figure 7 shows the results of these measurements. The two environments can cause similar amounts of interference on the TLB but our approach causes more LLC contentions. As a result, there is accordingly more bus traffic. Our tuning technique most likely has determined a more chaotic access pattern that can cause more cache misses on the Pi3.

It is easy to imagine that future work might tap into such performance counters as an objective function for tuning, providing higher confidence that tuned enemy programs more directly target their interference path.

V. Related Work

Multicore Interference: In most approaches, developers tune each resource-stressing benchmark for each specific SoC to detect interference patterns that are specific to the underlying microarchitecture of the system. Radojkovic et al. [24] were the first to utilise such techniques by deploying assembly code to measure multicore interference on real application workloads. They propose and evaluate a framework for quantifying the slowdown obtained during simultaneous execution by stressing a single shared resource at a time.

Nowotsch et al. [21] perform a similar experiment on a PowerPC-based processor platform and focus specifically on the memory system. The platform allows for different memory configurations and provides several methods for interference



(a) PI

(b) Joule

(c) Dragon

(d) Nano

(e) Banana

Fig. 6: Comparing the slowdowns from the Bechtel and Yun environment (BYE) [3] with the ones from our Tuned Hostile Environment (THE). Each ellipse represents one benchmark program. The width and height of each ellipse represent 95% confidence intervals in the slowdown measurements. Ellipses above the diagonal represent benchmarks where our method leads to larger slowdowns. The axis use a log scale. In the lower right corner we note the total number of cases when one method was statistically better than the other.

mitigation. Fernandez et al. [12] evaluate a multicore LEON-based processor and run experiments both with Linux and with RTEMS. Surprisingly, there are cases when the slowdown is even worse using the real-time OS.

While these approaches can yield impressive results, they require detailed knowledge of the underlying architecture. Fine-tuning requires significant engineering work and cannot be used to quickly eliminate unreliable contenders. In contrast, our black-box approach assumes only basic architectural knowledge and can be used to estimate interference quickly and automatically.

Multicore Timing Anomalies: Shah et al. [25] study timing anomalies in multicore processors and show how more aggressive co-existing applications can actually lead to faster execution times. Fernandez et al. [11] show how resource-stressing benchmarks may fail to produce *safe* bounds. Under heavy contention, arbitration policies for shared resources, such as round robin and first-in-first-out, can lead a PUT to suffer a delay that is not as severe as the potential worst case. This emphasises the difficulty of hand-crafting enemy programs and highlights the need for automatic methods such as ours.

Interference Mitigation: Several mitigation techniques for multicore interference have been proposed. First, prior works have examined potential possible interference paths, where contention for shared resources might impact program execution time [15], [22], [23], [4], [5]. Techniques have then been proposed for limiting this interference, either through hardware support (e.g. cache partitioning [19]) or invasive software modifications (e.g. bank-aware memory allocation and bandwidth reservation [34], [33]). However, even with these schemes, interference can still be substantial [29]. Therefore, a technique to validate, and probe the limits of, the mitigation method is still needed.

Limitations, Tuning and Microbenchmarks: Trying to derive empirically the worst-case multicore interference for a

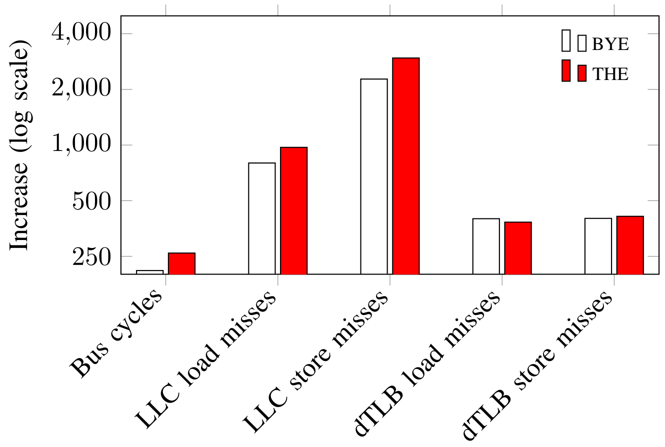


Fig. 7: Average relative increase in PMC value over all benchmarks on the Pi3.

task is arguably impossible. The work by Diaz et al. [17] has resulted in the use of contention models of specific processor resources based on information gathered from PMCs. These models can be used to compose delays due to contention. Griffin et al. [13] train a neural network to learn the relationship between interference and the effect of the PUT execution time. This approach is used to calculate an interference multiplier that can be applied to a previously calculated WCET without interference. Our work is complementary and can be used to cross-validate the results obtained using these approaches.

Tuning strategies have been used to optimise different computational aspects, with Ansel et al. [1] showing how such an approach can be used for a variety of domain-specific issues. Wegner et al. [30] use genetic algorithms to find the inputs that cause the longest or shortest execution time. Law et al. [18] use simulated annealing on a single-core processor to maximise code coverage and therefore obtain an estimate of the WCET. However, such automatic approaches have not been applied to multicore processors until now.

Microbenchmarks, with functionality similar to the enemy programs, have been successfully used for other purposes. Eklov et al. [9], [8] deploy such enemies to measure the latency and bandwidth sensitivity of a given application. Fang et al. [10] use a type of microbenchmark to determine microarchitectural details of a processor.

VI. Conclusions

We have presented (1) a reliable methodology to measure empirical multi-core interference, and (2) an auto-tuning framework, enabled by our reliable measurement approach, for finding effective parameters for enemy processes. We evaluated this method across a wide range of processors, both Arm and x86, using industry-standard benchmarks. Comparing the slowdowns caused by our auto-tuned enemies with hand-tuned enemies from prior work, we showed that our slowdowns are often (35 out of 105) more pronounced, and up to 3.8× larger. Thus, we have shown that this approach provides a straightforward, effective, and reliable technique for initial evaluations of multicore processors for tasks with real-time constraints.

TABLE V: Other enemy templates considered.

Shared resource targeted	Operation performed
Bus	Transfers between the CPU and RAM
Memory Thrashing	Random writes to main memory
Pipeline	Arithmetic operations
System	I/O operations

Acknowledgments

This work was supported by DSTL grant R1000115750: Multi-Core Microprocessor Test Harness, by the EPSRC IRIS Programme Grant (EP/R006865/1) and by the HiPEDS Grant EP/L016796/1. We thank our anonymous reviewers whose comments greatly improved the manuscript.

Appendix

A. Graveyard of enemies

Our approach depends on the quality of the enemy programs that we have chosen. Table V highlights other enemy templates that we have considered.

The bus and memory thrashing enemy programs were meant to similarly stress elements of the memory hierarchy. However, these were completely overshadowed by the memory enemy (shown in Table II) in terms of interference caused. For this reason, the second stage of our tuning process would always eliminate them. We therefore can see the strength of our tuning strategy to help us eliminate inefficient templates.

The pipeline enemy programs was inspired by previous work [24] but such an enemy is most likely effective when using a multi-threaded architecture with multiple virtual threads sharing the same physical core. Since we only focused on small embedded devices, which typically do not run many tasks simultaneously, this proved not to be the case. We decided to leave this exploration of this type of interference to future work.

The final enemy we considered was meant to stress the operating system by causing a significant number of operations on files. It created significant interference, considerably slowing down other processes. We felt that this was beyond the scope of the paper, since the interference caused was heavily influenced by the choice of operating system. Nevertheless, such an enemy would be interesting to study if we wished to evaluate the robustness of embedded operating systems.

It is up to the user to choose good enemy/victim pairs, and we have provided two such pairs in this paper. If they write a poor enemy or a poor victim then the technique will not find useful interference. The enemy/victim examples we present here are based on best efforts from prior work [3] that we have found to work well in practice.

References

[1] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *International Conference on Parallel Architectures and Compilation Techniques*, Edmonton, Canada, August 2014.

[2] Autobench2. *The EEMBC Autobench 2 Benchmark suite*, 2018. from https://www.eembc.org/autobench2//index.php.

[3] M. Bechtel and H. Yun. Denial-of-service attacks on shared cache in multicore: Analysis and prevention. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 357–367, April 2019.

[4] Guy Berthon, Marc Fumey, Xavier Jean, Helene Misson, Laurence Mutuel, and Didier Regis. White Paper on Issues Associated with Interference Applied to Multicore Processors. Technical report, Thales Avionics, 2733 South Crystal Drive, Suite 1200, Arlington, VA 22202, 2016.

[5] Vincent Brindejonc and Roger Anthony. Avoidance of Dysfunctional Behavior of Complex COTS Used in an Aeronautical Context. In *Lambda-Mu RAMS Conference*, Dijon, France, 2014.

[6] Coremark. *The EEMBC Coremark Benchmark suite*, 2009. from https://www.eembc.org/coremark//index.php.

[7] Augusto Born de Oliveira, Sebastian Fischmeister, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Why you should care about quantile regression. *SIGARCH Comput. Archit. News*, 41(1):207–218, March 2013.

[8] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten. Cache pirating: Measuring the curse of the shared cache. In *2011 International Conference on Parallel Processing*, pages 165–175, Sep. 2011.

[9] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten. Bandwidth bandit: Understanding memory contention. In *2012 IEEE International Symposium on Performance Analysis of Systems Software*, pages 116–117, April 2012.

[10] Zhenman Fang, Sanyam Mehta, Pen-Chung Yew, Antonia Zhai, James Greensky, Gautham Beeraka, and Binyu Zang. Measuring microarchitectural details of multi- and many-core memory systems through microbenchmarking. *ACM Trans. Archit. Code Optim.*, 11(4):55:1–55:26, January 2015.

[11] G. Fernandez, J. Jalle, J. Abella, E. Quiones, T. Vardanega, and F. J. Cazorla. Computing safe contention bounds for multicore resources with round-robin and FIFO arbitration. *IEEE Transactions on Computers*, 66(4):586–600, April 2017.

[12] Mikel Fernández, Roberto Gioiosa, Eduardo Quiñones, Luca Fossati, Marco Zulianello, and Francisco J. Cazorla. Assessing the suitability of the NGMP multi-core processor in the space domain. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '12, pages 175–184, New York, NY, USA, 2012. ACM.

[13] David Griffin, Benjamin Lesage, Iain Bate, Frank Soboczenski, and Robert I. Davis. Forecast-based interference: Modelling multicore interference from observable factors. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, RTNS '17, pages 198–207, New York, NY, USA, 2017. ACM.

[14] Gerald J. Hahn, William Q. Meeker, and Luis A. Escobar. *Statistical Intervals: A Guide for Practitioners*. Wiley Publishing, 2nd edition, 2014.

[15] L. M. Kinnan. Use of multicore processors in avionics systems and its potential impact on implementation and certification. In *2009 IEEE/AIAA 28th Digital Avionics Systems Conference*, pages 1.E.4–1–1.E.4–6, Oct 2009.

[16] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[17] L. Kosmidis, R. Vargas, D. Morales, E. Quiones, J. Abella, and F. J. Cazorla. TASA: Toolchain-agnostic static software randomisation for critical real-time systems. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, Nov 2016.

[18] Stephen Law and Iain Bate. Achieving Appropriate Test Coverage for Reliable Measurement-Based Timing Analysis. *Proceedings - Euromicro Conference on Real-Time Systems*, 2016-Augus:189–199, 2016.

[19] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Enabling software management for multicore caches with a lightweight hardware support. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, Nov 2009.

[20] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, 18(1):50–60, 1947.

[21] J. Nowotsch and M. Paulitsch. Leveraging multi-core computing architectures in avionics. In *2012 Ninth European Dependable Computing Conference*, pages 132–143, May 2012.

[22] P. Parkinson. The challenges of developing embedded real-time aerospace applications on next generation multi-core processors. In *Aviation Electronics Europe, Munich, Germany*, April 2016.

[23] P. Parkinson. Update on using multicore processors with a commercial arinc 653 implementation. In *Aviation Electronics Europe, Munich, Germany,e*, April 2017.

[24] Petar Radojković, Sylvain Girbal, Arnaud Grasset, Eduardo Quiñones, Sami Yehia, and Francisco J. Cazorla. On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments. *ACM Trans. Archit. Code Optim.*, 8(4):34:1–34:25, January 2012.

[25] H. Shah, K. Huang, and A. Knoll. Timing anomalies in multi-core architectures due to the interference on the shared resources. In *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 708–713, Jan 2014.

[26] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 2951–2959. Curran Associates, Inc., 2012.

[27] Tyler Sorensen and Alastair F. Donaldson. Exposing errors related to weak memory in GPU applications. *SIGPLAN Not.*, 51(6):100–113, June 2016.

[28] P. K. Valsan, H. Yun, and F. Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, April 2016.

[29] Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. Addressing isolation challenges of non-blocking caches for multicore real-time systems. *Real-Time Systems*, 53(5):673–708, Sep 2017.

[30] Joachim Wegener, Harmen Sthamer, Bryan F. Jones, and David E. Eyres. Testing real-time systems using genetic algorithms. *Software Quality Journal*, 6(2):127–135, Jun 1997.

[31] Reinhard Wilhelm. Mixed Feelings About Mixed Criticality (Invited Paper). In Florian Brandner, editor, *18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018)*, volume 63 of *OpenAccess Series in Informatics (OASIcs)*, pages 1:1–1:9, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[32] Clark Williams. *Latency Evaluation framework*, 2018. Available online https://git.kernel.org/pub/scm/linux/kernel/git/clrkwllms/rt-tests.git/.

[33] H. Yun, R. Mancuso, Z. P. Wu, and R. Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 155–166, April 2014.

[34] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, April 2013.