Revisiting Migration Overheads in Real-Time Systems: One Look at Not-So-Uniform Platforms

Phillip Raffeck, Wolfgang Schröder-Preikschat
Friedrich-Alexander-Universität Erlangen-Nürnberg
raffeck@cs.fau.de, wosch@cs.fau.de

Peter Ulbrich
Technische Universität Dortmund
peter.ulbrich@tu-dortmund.de

***Abstract*—Dynamic migration of tasks between cores is nowadays one of the standard mechanisms of operating systems to exploit multi-core systems. However, migration is practically not used in real-time settings. This is due to the unpredictability of the associated costs and the resulting pessimistic overapproximations. Existing approaches typically rely either on a predictable, static partitioning of tasks or assume uniform costs for all cores; conceptually, migration does not differ from preemption in the latter. However, non-unified memory architecture (NUMA) and NUMA-like embedded hardware platforms are increasingly widespread. Here, intuitively, migration should be more costly than preemption, but the degree is uncertain.**

This paper aims to shed more light on two key influencing factors: (1) the variance of the elementary costs of the hardware and (2) the type and scope of the affected data of a task at the time of migration, the working set. We approach this challenge by deeper investigating application benchmarks, revisiting existing cost experiments, and bringing them to new platforms with not-so-uniform memory architectures. Our results indicate that migration differs from preemption in many relevant cases and thus requires special consideration to incorporate the associated overheads precisely into worst-case analyses.

I. Introduction

A static allocation of workloads to cores can lead to the well-known Dhall effect [1] and is therefore generally considered detrimental to utilization in multi-core settings. Nowadays, we are used to dynamically distributing workloads conveniently between cores in a multi-core system, for example, by migrating processes, containers, or even complete virtual machines. The operating system's primary challenge is to provide transparent and efficient migration mechanisms that foster CPU utilization. However, things are significantly more complicated in the domain of (hard) real-time systems. For example, although scheduling theorists have embraced transparent task migration at the instruction level, it is rarely put into practice in real-world applications. The primary reason for this is that the effects on the temporal behavior are much more complex to predict—a decades-old problem [2]. The *worst-case execution times* (WCET) of application, and operating system functionality are decisive for verifiable scheduling and typically inferred by static analysis techniques. However, the ease and difficulty of such timing analysis are dictated by the given hardware and software's predictability (and their analyzability). Even slight variabilities in execution costs can cause excessive pessimism in WCET estimates and thus jeopardize schedulability and the desired improvement in overall utilization. Conceptually, two main influencing factors can be distinguished: (1) Variable execution costs of the elementary operations caused by memory-access related latencies, i.e., the WCET varies with core allocation. (2) Migration overheads induced by the program structure of the task to be migrated. In particular, these are determined by the working set, i.e., the live part of a program's resident set, which must be transferred between cores.

We first tackled the second factor by static analysis of tasks to determine their resident (RSS) and *working-set sizes* (WSS) in previous works [3], [4]. As a result, we could reduce analysis pessimism by identifying particularly advantageous migration points with low maximum cost. Furthermore, by compiler-supported slicing of the tasks at these points, we obtained smaller jobs that are easier to allocate and schedule, fostering predictable multi-core scheduling.

Challenges and Contribution

Our previous and ongoing work is based on two key assumptions: First, the working-set size varies within the program execution so that there are beneficial migration points to identify. Second, we assume that the hardware exhibits variable access and thus elementary costs on different cores. While we could demonstrate these basic assumptions for a static allocation using core-local memories [3], the generalizability remained an open question.

On the one hand, this issue relates to the evolution of the size of the resident and working set over time in typical applications. Does its size vary within programs and during their execution?

On the other hand, the more challenging question is the variable execution cost on hardware platforms with advanced memory architectures. For example, in their study of preemption and migration delays, Bastoni et al. [5] found these converged with a task sufficiently long preempted. However, they ran their experiments on a *unified memory architecture* (UMA) machine. Because the memory access latencies are equal to all cores, the cache state (i.e., hotness) determines the access latencies. The situation is unclear in systems with less uniform memory characteristics (e.g., *non-unified memory architecture*, NUMA), where certain memory regions may not be accessible from all cores or only with significant overhead. Because of potential data transfers or memory-access overheads for the

migrated task, migration should intuitively be more costly than preemption, but the degree is uncertain.

This paper aims to shed more light on these questions by deeper investigating application benchmarks, revisiting existing cost experiments, and bringing them to new platforms. This will enable us to identify potential scenarios where consideration of migration overhead is beneficial or necessary and clear up misconceptions or uncertainties about the overhead associated with migration.

The paper provides the following contributions: (1) A compile-time analysis of working-set sizes of different benchmarks typical for the domain of real-time systems. (2) The reproduction of previously published results for preemption and migration delays in real-time systems on UMA platforms. (3) Bringing these experiments to a broader spectrum of platforms and memory architectures, namely two x86 platforms with 4 and 8 NUMA domains. (4) A more detailed insight into embedded systems with NUMA-like characteristics by comparable measurements on a typical real-time platform.

II. Approach

We aim to augment the existing data on migration overheads in a two-pronged fashion. First, we investigate the influence of migration on target platforms both with and without NUMA characteristics. Starting with a reproduction of previous results [5], our study provides a more in-depth evaluation of migration delays. Further, we analyze benchmarks at compile time to get a better understanding of how WSSs grow and shrink over the lifetime of a task.

A. Measurements

To obtain a broad overview of the costs and implications of migration, we observed task execution and memory access behavior on a range of platforms. Depending on the platform, we used different observation methods, which we briefly summarize in the following:

On all platforms, we measured memory overheads in a direct approach by recording memory access times to different memory regions and NUMA domains.

On platforms where PREEMPT_RT Linux [6] is readily available, we additionally employed an indirect method to observe migration overheads. Using the ftrace [7] functionality of the Linux kernel, we examined the runtime of a measurement task[1] embedded in specifically crafted task systems consisting of an interference task and multiple blocker tasks. By tuning the period and execution time of the interference task, we were able to trigger preemptions and migrations in the measurement task, which become visible as scheduling events in the ftrace output. The measurement task's execution times derived from these scheduling events subsequentially allowed us to conclude the overhead of preemptions and migrations.

On the embedded platform, we additionally observed the overhead incurred by placing a task's data (e.g., its stack, relevant parts of data sections) in different domains in the memory hierarchy. This setup mimics the loss of core locality due to migration, which enabled us to study two strategies: control-flow-only migration (i.e., data resides in core-local memories) and storing data in shared memory only. We then assessed the potential impacts of such strategies on task execution from the observed overheads.

[1]The measurement task is the subject of the measurement. However, in some experiments, it is also instrumented for measuring execution time.

Name	CPU	Cores	NUMA Domains
M1	Intel i7-2600	4	1
M2	AMD Opteron 6180	48	8
M3	Intel Xenon E7-4830	48	4

Table I: Overview of the x86 machines used for measurements.

B. Analysis Approach

To better understand the variation of the WSS of tasks over their lifetime, we employed a variation of the analysis routine described in [3]. Using a custom LLVM-based [8] compiler [9], [10], we determined the live-data set at each instruction for our benchmarks during compilation. Contrary to our previous work, we were not interested in identifying beneficial points in close vicinity of a target WCET, but instead in the distribution and evolution of the WSS at all points in the execution of the task.

This approach allowed us to evaluate the impact of worst-case WSS estimates on the task execution, as we gained an overview of how often these worst cases actually occur. This, in turn, provided insight into the degree of overapproximation that is introduced by pessimistic worst-case estimates of migration-induced overheads. As our analysis is performed during the compilation stage, the size estimates are based on LLVM data types in the granularity of single bits. Currently, the analysis only considers stack-based dynamic memory allocation, which we deem sufficient for the target domain of (hard) real-time systems. Even though analyzing the representative benchmarks used in our evaluation did not require it, an extension to heap-based memory management is feasible with restrictions on idiomatic C (alias problem) [11]–[13]. In sum, our approach provided valuable information to assess the impact of migration overheads.

III. Observation of NUMA-Effects

We performed experiments on various platforms to cover the characteristics of different memory architectures and their influence on migration overheads. First, as a powerful embedded platform representative, we opted for the Infineon AURIX platform, which is widely used in safety-critical automotive applications (e.g., engine and body control, collision avoidance systems). Specifically, we used the 6-core AURIX TriCore TC397XE [14] for our experiments. The platform features a diverse memory hierarchy, including core-local scratchpads and multiple levels of CPU-local (DLMUx) and domain-local memory (LMUx), as well as global extended memory (EMEM). The shared memory regions are mirrored to two different address ranges to allow cached and non-cached access. The core-local scratchpads are accessible from all cores, albeit at the cost of higher latencies. Cores 0-3 and 4-5, as well

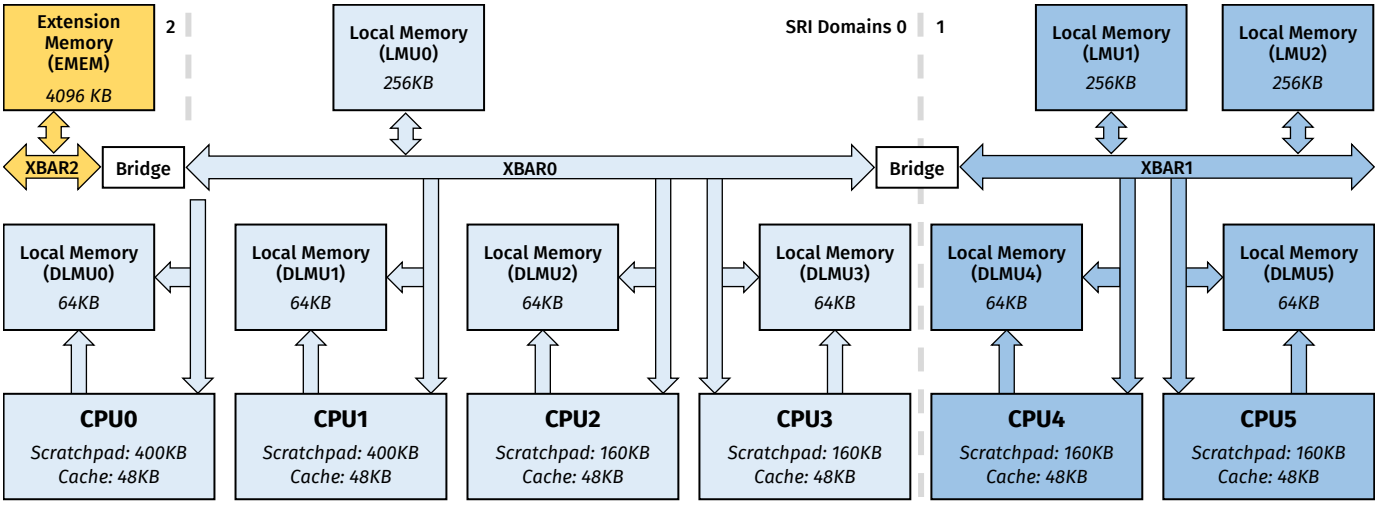


Figure 1: Block diagram of the TC3X microcontroller memory architecture. It features multiple levels with varying access latencies: from core-local scratchpad, over (domain) local memory (DLMU, LMU) up to extended memory (EMEM). Domains are marked by color, linked by the system resource interconnect (SRI, i.e., crossbar), and accessible via bridges.

as the extended memory, are connected to different crossbars (XBARx), called system resource interconnect (SRI), constituting dedicated memory domains. Figure 1 outlines the components of the memory hierarchy relevant to this work.

In the following, we first discuss the results on the AURIX platform before moving on to the second class of systems: three x86 machines with different core counts and NUMA characteristics, ranging from common desktop CPUs to many-core platforms. Table I provides an overview of the respective hardware details of the used CPUs.

A. AURIX embedded platform experimental results

Experiments were performed on an Infineon AURIX TC397XE [14] platform with the help of a Lauterbach PowerDebug [15]. Using this hardware debugger allowed us to record instruction traces in a non-intrusive manner by on-chip tracing. One limitation of this, however, is that the measurements are limited by the 1 MB trace buffer. By recording just the start and end times of the relevant code sections for the experiments, we ensured that the length of the benchmarks did not become an issue of said limited buffer size.

In the first experiment, we examined access times from one core into different parts of the memory hierarchy of the TC397XE platform. All memory regions, including scratchpads of other cores, are accessible from all cores, albeit with different latencies. To quantify these, we captured execution traces for transferring a total of 1 KiB, 8 KiB, 16 KiB and 32 KiB of memory to core 0 from the different memory regions and domains to cover all the various possible access routes through the memory interconnects showcased in Figure 1. The transfer was performed at word granularity, with the next word to transfer chosen randomly to avoid prefetching effects of linear memory accesses. In particular, we evaluated accesses from four memory regions: First, the core-local scratchpad of core 0 as the typical storage for task data. Additionally, one of the shared domain-local memory regions close to core 0, which is (D)LMU0 in Figure 1, in both the cached (LMU0) and non-cached (LMU0 NC) variant. We consider only the non-cached access variant as relevant, as evaluating cache-coherency mechanisms is outside the scope of this paper. Lastly, we used the scratchpad of core 4 as a remote memory region, constituting the worst-case scenario as it is only accessible for core 0 through the crossbar bridge and subsequently via core 4's interface.

Figure 2 gives a box plot of the determined latencies normalized to access times per word (i.e., 32 bit). Median values are marked by circles, outliers by diamonds. There are three distinct groups visible: (1) core-local and cached access to shared memory, (2) non-cached access to shared memory, and (3) access to the scratchpad of a remote core.

The measurements suggest that the execution time of a task may vary significantly on an embedded platform with NUMA-like characteristics depending on where exactly the required data resides. Especially the near doubling of access times between core-local and remote scratchpad accesses stuck out.

Considering these numbers with migration in mind, they indicate potentially significant overheads, either because the WSS of a task has to be transferred or because the execution time of a task changed due to the higher access times of memory accesses, which became remote accesses after the migration. Depending on the cache-coherency requirements and available mechanisms, moving data to shared memory is a no viable solution, as indicated by the increased access times for non-cached access to shared memory.

For the second experiment, we evaluated the execution times of tasks derived from the TACLeBench benchmark suite [16] in a simulated migration scenario. We mapped the stack and data regions used by the examined task to different memory regions during linking. On the one hand, this simulated a complete task execution after a migration to a

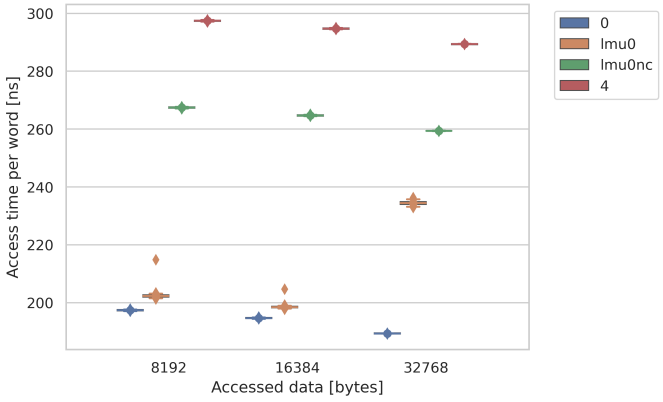


Figure 2: Access times for different parts of the memory map

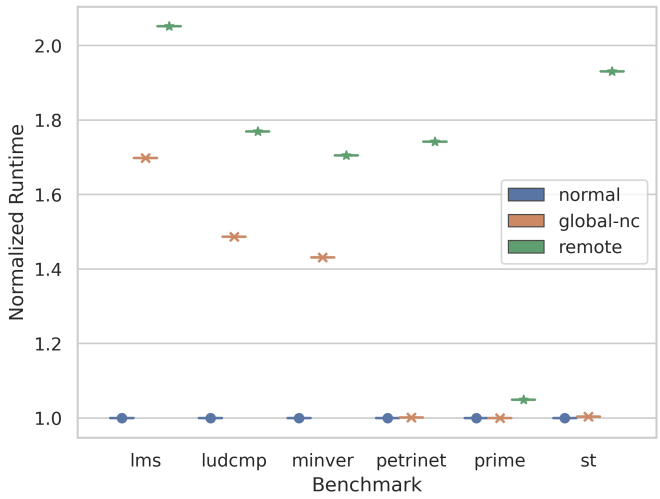


Figure 3: Runtimes for TacleBench benchmarks with their data residing in different parts of the memory map

different core. On the other hand, this allowed us to compare further the effects of attempts to circumvent problems coming with migration by moving data to global shared memory.

Figure 3 showcases execution times of several benchmarks in three different memory configurations as a boxplot. The data used by the tasks lay either in the scratchpad of core 0 (normal case), in the non-cached shared memory (global-nc), or in the scratchpad of core 4 (remote). The median values of the measurements are marked by a circle (normal), a cross (global-nc), and a star (remote), respectively. For easier comparison, the execution times are normalized to the standard case.

Again, we noticed significant differences in the execution times depending on where the task data resides in memory. These are more or less pronounced depending on the specific benchmark, its memory footprint, and WSS access patterns.

Two conclusions can be drawn from the results, albeit still depending on the concrete nature of the tasks:

(1) Operating entirely in shared memory may come with significant overheads. This signifies the reality of scenarios where migration is beneficial or necessary.

(2) Depending on the target core, migration comes with

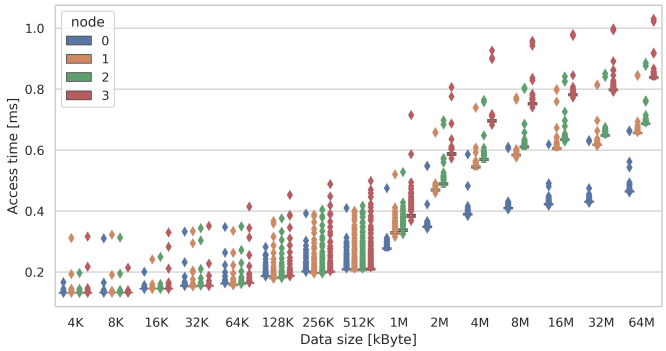


Figure 4: Memory access times from different NUMA domains on `M2`. Benchmark run on domain 0.

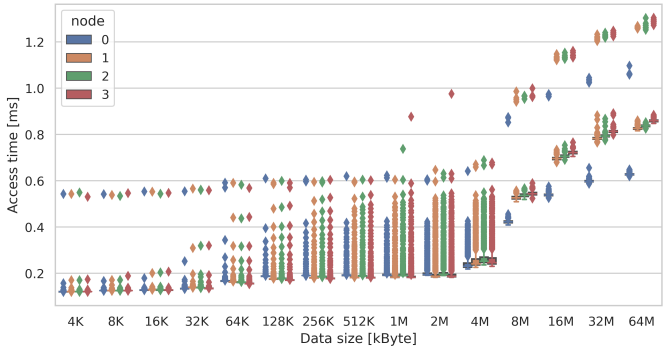


Figure 5: Memory access times from different NUMA domains on `M3`. Benchmark run on domain 0.

substantial costs. Thus, we mandate predictable migration with precise WSS estimation to manage the overhead.

B. x86 NUMA platforms experimental results

On the three larger platforms, we evaluated memory access times by performing 4096 reads and writes at random indices of an increasingly larger array and controlling the memory allocation of said array via `numactl`[2]. Figures 4 and 5 display the measured access times as boxplots. For better readability, only 4 of the 8 NUMA domains of `M2` are displayed, as the results are equivalent for the other domains. On both machines, we can see larger access times from across NUMA domains for larger array sizes.

Additionally, we ran PREEMPT_RT Linux [6] to leverage the Linux tracing functionality [7]. For a finer resolution of the results, we splitted the existing `sched_switch` event in two, `sched_switch_on` and `sched_switch_off`. Evaluations were performed using the `nop` tracer, the `TSC` as the clock source, and with only our scheduling events enabled. Real-time throttling was disabled during the measurements.

The task system under observation comprised one high-priority measurement task and one interference task for each processor core, which perpetually accessed a WSS the same size as that of the measurement task. We experimented with different microbenchmarks as measurement

[2]https://github.com/numactl/numactl

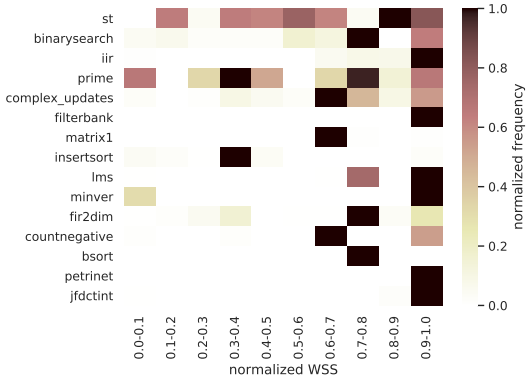


Figure 6: WSS distribution of TACLeBench benchmarks

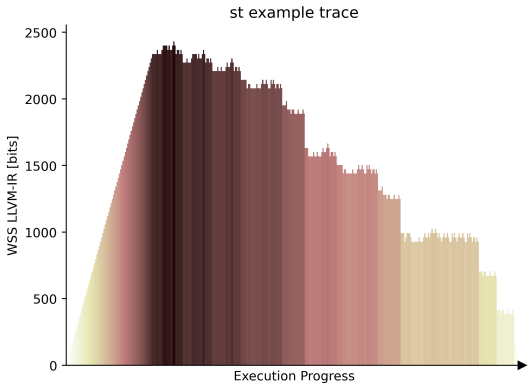


Figure 7: WSS distribution over an exemplary trace of the benchmark `st`

tasks: one processor-bound (`CPU`) and one memory-bound benchmark (`MEM`). The processor-bound benchmark calculates primes up to two alternating limits. The memory-bound benchmark moves sequentially through a working set of predefined size reading bytes with a step width of 32 B, each read immediately followed by a write to the following byte. We use 2 KiB as a small, 32 KiB as a medium, and 786 KiB as a large WSS. In comparison to the results of our analysis of embedded benchmarks (see Section IV, these constitute rather large working-set sizes. To observe worst-case memory effects, the measurement task switched CPU every few iterations as indicated by the different colors in Figures 8 to 12.

The traced execution times of the measurement task during these experiments are displayed in Figures 8 to 11. For the CPU benchmark, we identified two execution-time clusters on all machines, which fit the code's behavior. `M3` shows a pattern of execution time spikes followed by regions of lower execution times. The effect is better observable in a zoomed-in comparison (see Figure 12): For the MEM benchmark, higher costs are visible at switches to a new processor, followed by faster executions as the caches warm up. At the beginning and the crossing of NUMA domains, distinct spikes are seeable, indicating higher overheads for crossing NUMA domains. The comparison with the execution times of the CPU benchmark shows that memory access latencies are responsible for the observed pattern. Additionally, we evaluated a WSS of 8 MiB for `M3` as the size the access experiments (Figure 5) where NUMA effects become more prevalent. In comparison with Figures 11c and 12b, we can identify higher cost after switching NUMA domains but not after switching cores, indicating that the influence of caches declines at such huge WSSs, while the NUMA influence remains. For `M1`, no clear effect distinguishes preemption and migration, reaffirming the observations of Bastoni et al. [5]. On `M2`, the effect is less prevalent, indicating that NUMA architectures do not necessarily come with detrimental overheads.

In summary, these results suggest that dissimilarities between preemption and migration are likely more prevalent in NUMA systems, requiring the explicit consideration of migration for sound and precise WCET estimates.

IV. Distribution of Working-Set Sizes

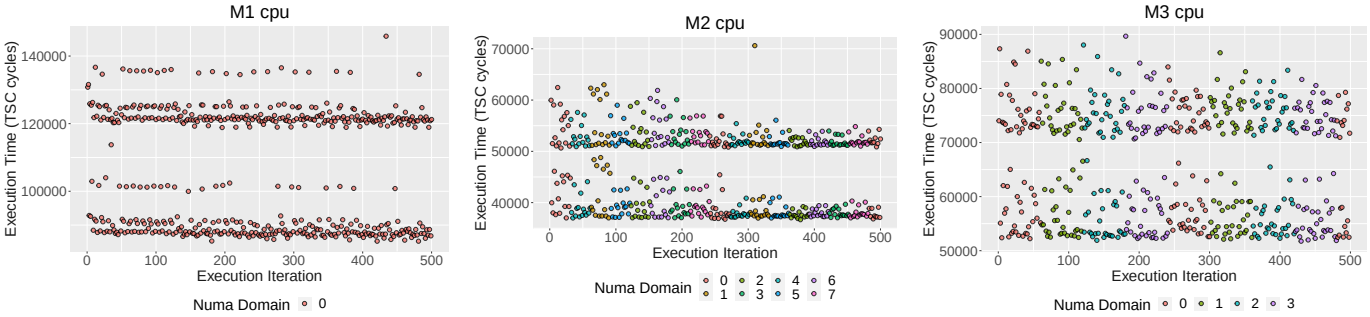
The results of Section III imply that restricting migration to smaller WSSs is beneficial. In a second step, we studied the evolution of WSSs over the lifetime of a task. Therefore, we applied the analysis described in Section II-B on TACLeBench benchmarks [16], which serve us as representatives for typical application patterns.

Figure 6 shows a heat map of the resulting WSS distribution. For easier comparison, the results are normalized twice: first, for each benchmark (i.e., each row), the WSS is normalized to the interval of its respective minimum and maximum size. Second, the occurrence frequency is normalized in the same way. Each column represents a tenth of the WSS interval, while the color of each field indicates how many of the observed WSSs for the benchmark lie within the interval of the column; darker colors denote a higher frequency. We found all working sets in absolute numbers to be smaller than 10 KiB by our analysis.

The distribution shows no clear trend across all benchmarks but rather a high degree of variation. While some benchmarks (e.g., filterbank, jfdctint, petrinet) exhibit a WSS equivalent to near the worst-case estimate most of the time, others (e.g., prime, st) show a wider distribution and several peaks over different WSS ranges.

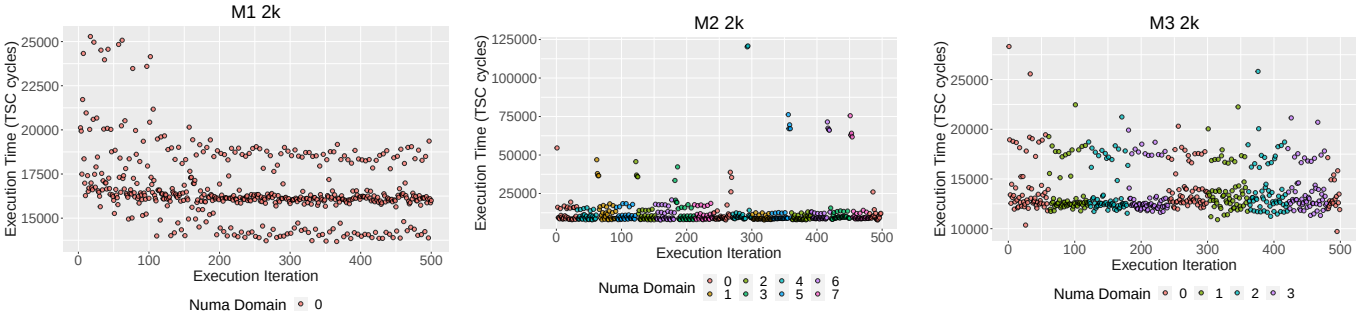
As an example, Figure 7 shows the progression of the WSS over one possible execution trace of the `st` benchmark. Every bar represents the WSS at one instruction. Instructions are ordered from left to right by their time of occurrence in the trace. Note that the selected trace does not necessarily represent the worst-case execution but rather just one potential execution. As the figure shows, there are notable differences in the WSS throughout the execution, indicating that the data to be transferred in the case of migration strongly depends on the exact point in time the migration takes place.

Two conclusions can be drawn from these analysis: (1) Whether the worst-case WSS is representative of a task execution strongly depends on the nature of the task, as it may as well access only a far smaller WSS most of its time. (2) Predictable migration is essential to improve worst-case



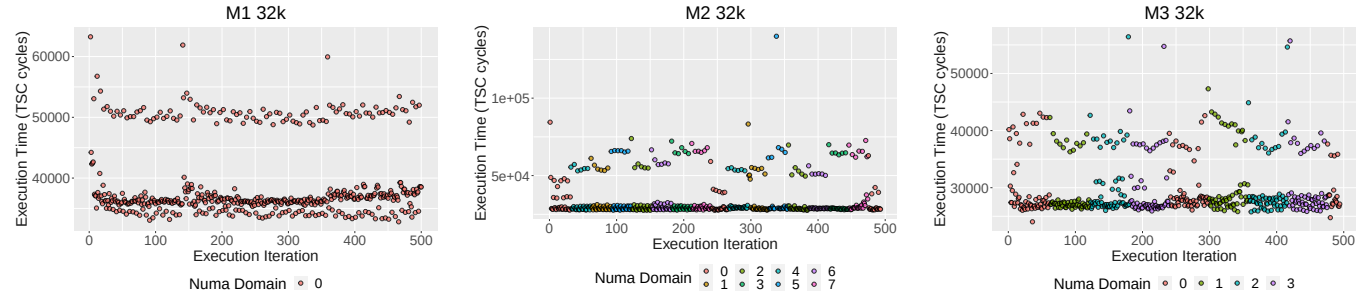
(a) M1 (b) M2 (c) M3

Figure 8: Trace results for CPU.



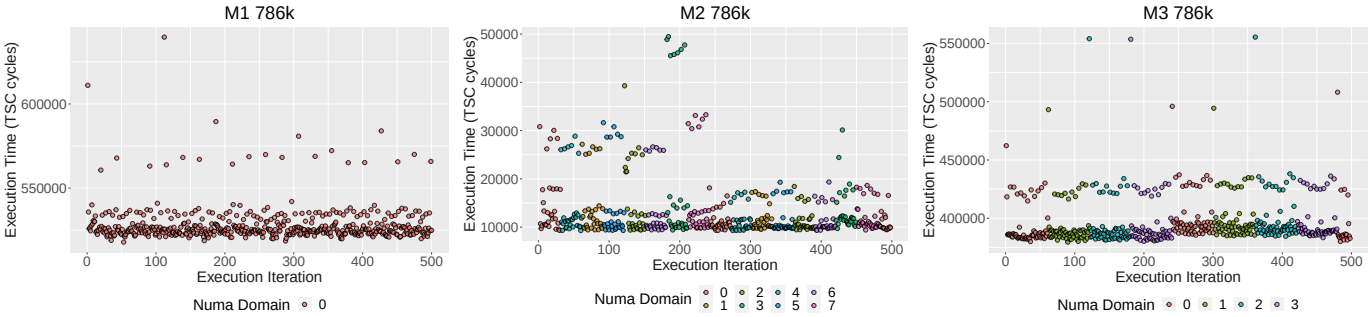
(a) M1 (b) M2 (c) M3

Figure 9: Trace results for MEM with a WSS of 2k.



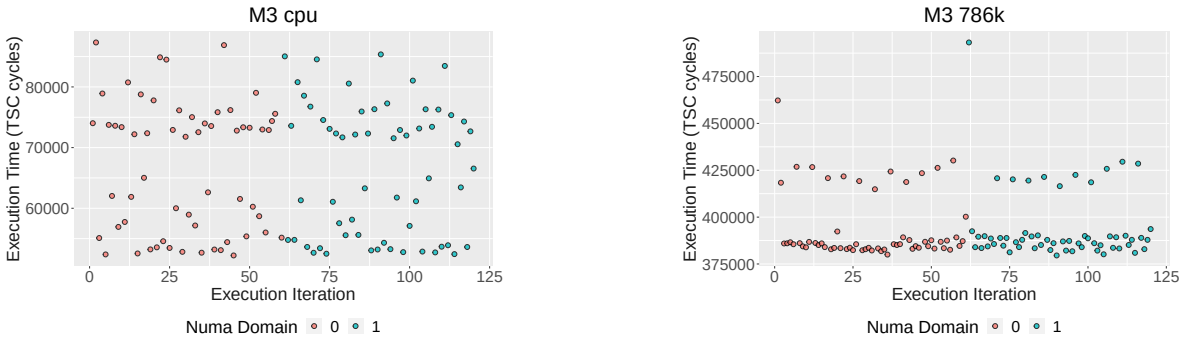
(a) M1 (b) M2 (c) M3

Figure 10: Trace results for MEM with a WSS of 32k.



(a) M1 (b) M2 (c) M3

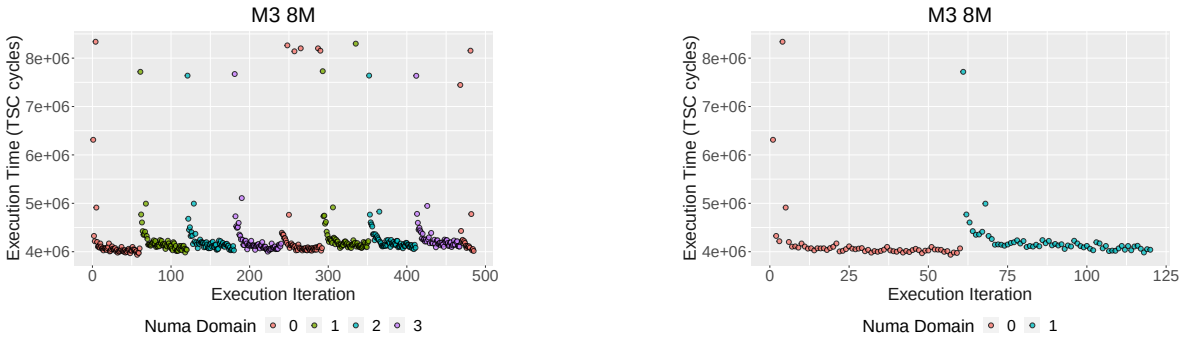
Figure 11: Trace results for MEM with a WSS of 786k.



(a) CPU

(b) MEM 786k

Figure 12: Zoomed version of the trace results on `M3`.



(a) Trace results.

(b) Zoomed version of the trace results.

Figure 13: Trace results for MEM with a WSS of 8M on `M3`.

analyses by avoiding unnecessary pessimistic WSS estimates. Knowledge of the exact point in time a migration happens facilitates precise WSS and, hence, overhead estimation.

V. Related Work

Since the beginning of multitasking, working-set estimation has been a research topic with a large body of work [17]–[19], especially in (virtual) memory management. Brown et al. [20] presented a technique for working-set prediction to make thread migration more efficient. They highlight the need to precisely identify the actual current working set to prevent unnecessary data transfers. In real-time systems, Calandrino et al. [21] and Bastoni et al. [5] identified the WSSs to impact the worst-case timing analysis significantly. Beyond these general investigations, we have studied the specific properties of typical application benchmarks in this work.

Likewise, preemption and migration costs are a vast area of research. For example, Bastoni et al. [5] measured preemption and migration delays on a 24-core UMA machine. They observed no significant differences between preemptions and migrations in systems under load, as with increasing preemption length, cache affinity is lost either way completely. Contrary, in a comparable experiment, Calandrino et al. [21] observed worst-case migration costs to be higher due to forced data invalidation and cache-coherency overheads. Work on cache-related preemption delays (CRPD) [22], [23] aims to determine the effects of preemption more accurately or minimize them systematically. However, we are not aware of any work that considers migration between NUMA domains.

VI. Conclusion & Outlook

Using the tracing capabilities of PREEMPT_RT Linux, we were able to validate previous results for preemption and migration overheads and extend them to the broader range of NUMA and embedded NUMA-like platforms. Our results substantiate the general intuition that migration and preemption deserve nuanced consideration in such scenarios, as the former is associated with more uncertainties and costs.

Further, static analysis of representative application benchmarks indicates that the (worst-case) resident set is an extensive overapproximation of the actual working set for most of the execution. Migration at predictable points in the execution, thus, helps to avoid unnecessary pessimism, as the WSS can be determined more precisely section by section.

Overall, we conclude that, generally, migration cannot be treated like preemption. Especially on NUMA-like systems, migration requires special consideration to precisely determine and incorporate overheads in the system design.

In previous work [4], we outlined possible approaches to enable predictable migration with known overheads by static and dynamic scheduling. The results presented in this paper will help us to refine these approaches and put them into practice.

ACKNOWLEDGMENT

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project numbers 146371743;198891422.

REFERENCES

[1] S. K. Dhall and C. L. Liu, "On a Real-time Scheduling Problem," *Operations Research*, vol. 26, no. 1, pp. 127–140, 1978.

[2] D. S. Milojičić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process migration," *ACM Computing Surveys*, vol. 32, no. 3, pp. 241–299, 2000.

[3] T. Klaus, P. Ulbrich, P. Raffeck, B. Frank, L. Wernet, M. R. von Onciul, and W. Schröder-Preikschat, "Boosting Job-Level Migration by Static Analysis (Best Paper Award)," in *Proc. of the 15th Intl. Work. on Operating Systems Platforms for Embedded Real-Time Applications*, 2019, pp. 33–44.

[4] P. Raffeck, P. Ulbrich, and W. Schröder-Preikschat, "Work-in-progress: Migration hints in real-time operating systems," in *Proc. of the 40th IEEE Intl. Real-Time Systems Symp.* IEEE, 2019, pp. 528–531.

[5] A. Bastoni, B. Brandenburg, and J. Anderson, "Cache-related Preemption and Migration Delays: Empirical Approximation and Impact on Schedulability," in *Proc. of the 6th Intl. Work. on Operating Systems Platforms for Embedded Real-Time Applications*, 2010, pp. 17–22.

[6] Real-time linux. [Online]. Available: https://wiki.linuxfoundation.org/realtime/start

[7] Ftrace - function tracer. [Online]. Available: https://www.kernel.org/doc/Documentation/trace/ftrace.txt

[8] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. of the Intl. Symp. on Code Generation and Optimization*, Washington, DC, USA, 2004, pp. 75–86.

[9] F. Scheler and W. Schröder-Preikschat, "The Real-time Systems Compiler: Migrating Event-triggered Systems to Time-triggered Systems," *Software: Practice and Experience*, vol. 41, no. 12, pp. 1491–1515, 2011.

[10] F. Franzmann, T. Klaus, P. Ulbrich, P. Deinhardt, B. Steffes, F. Scheler, and W. Schröder-Preikschat, "From intent to effect: Tool-based generation of time-triggered real-time systems on multi-core processors," in *Proc. of the 19th IEEE Intl. Symp. on OO Real-Time Distributed Computing*. Washington, DC, USA: IEEE, May 2016, pp. 134–141.

[11] M. Stilkerich, J. Schedel, P. Ulbrich, W. Schröder-Preikschat, and D. Lohmann, "Escaping the bonds of the legacy: Step-wise migration to a type-safe language in safety-critical embedded systems," in *Proc. of the 14th IEEE Intl. Symp. on OO Real-Time Distributed Computing*, G. Karsai, A. Polze, D.-H. Kim, and W. Steiner, Eds. IEEE, Mar. 2011, pp. 163–170.

[12] I. Stilkerich, C. Lang, C. Erhardt, and M. Stilkerich, "A practical getaway: Applications of escape analysis in embedded real-time systems," in *Proc. of the 14th ACM SIGPLAN/SIGBED Conf. on Languages, Compilers and Tools for Embedded Systems*, 2015, pp. 1–11.

[13] C. Lang and I. Stilkerich, "Design and implementation of an escape analysis in the context of safety-critical embedded systems," *ACM Trans. on Embedded Computing Systems*, vol. 19, no. 1, 2020.

[14] *Aurix Tc3xx User Manual Part 1*, Infineon, 2020, v1.6.0. [Online]. Available: https://www.infineon.com/dgdl/Infineon-AURIX_TC3xx_Part1-UserManual-v01_00-EN.pdf?fileId=5546d462712ef9b701717d3605221d96

[15] Lauterbach power debug interface usb3. [Online]. Available: https://www.lauterbach.com/powerdebugusb3.html

[16] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Wägemann, and S. Wegener, "TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research," in *Proc. of the 16th Intl. Work. on Worst-Case Execution Time Analysis*, ser. OpenAccess Series in Informatics (OASIcs), M. Schoeberl, Ed., vol. 55. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, pp. 2:1–2:10.

[17] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 10th ed. John Wiley & Sons, 2012.

[18] P. Denning, "Working sets past and present," *IEEE Trans. on Software Engineering*, vol. SE-6, no. 1, pp. 64–84, 1980.

[19] P. Bryant, "Predicting working set sizes," *IBM Journal of Research and Development*, vol. 19, no. 3, pp. 221–229, 1975.

[20] J. A. Brown, L. Porter, and D. M. Tullsen, "Fast thread migration via cache working set prediction," in *IEEE 17th Intl. Symp. on High Performance Computer Architecture*, 2011, pp. 193–204.

[21] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson, "LITMUSˆRT : A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers," in *Proc. of the 27th IEEE Intl. Real-Time Systems Symp.*, Dec. 2006, pp. 111–126.

[22] L. Ju, S. Chakraborty, and A. Roychoudhury, "Accounting for cache-related preemption delay in dynamic priority schedulability analysis," in *2007 Design, Automation Test in Europe Conf. Exhibition*, 2007, pp. 1–6.

[23] R. Mancuso, H. Yun, and I. Puaut, "Impact of DM-LRU on WCET: a Static Analysis Approach," in *31th Euromicro Conf. on Real-Time Systems*, ser. Leibniz Intl. Proc. in Informatics (LIPIcs), S. Quinton, Ed., vol. 107. Stuttgart, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, conference, pp. 17:1–17:25.