CARAT: Context-Aware Runtime Adaptive Task Migration for Multi Core Architectures

Janmartin Jahn†, M. A. Al Faruque, and Jörg Henkel
Karlsruhe Institute of Technology, Chair for Embedded Systems, Karlsruhe, Germany
{jahn, mohammad.faruque, henkel}@kit.edu

***Abstract*—Multi core architectures that are built to reap performance and energy efficiency benefits from the parallel execution of applications often employ runtime adaptive techniques in order to achieve, among others, load balancing, dynamic thermal management, and to enhance the reliability of a system. Typically, such runtime adaptation in the system level requires the ability to quickly and consistently migrate a task from one core to another. For distributed memory architectures, the policy for transferring the task context between source and destination cores is of vital importance to the performance and to the successful operation of the system. As its performance is negatively correlated with the communication overhead, energy consumption and the dissipated heat, task migration needs to be runtime adaptive to account for the system load, chip temperature, or battery capacity. This work presents a novel context-aware runtime adaptive task migration mechanism (CARAT) that reduces the task migration latency by 93.12%, 97.03% and 100% compared to three state-of-the-art mechanisms and allows to control the maximum migration delay and the performance overhead tradeoff at runtime. This novel mechanism is built on an in-depth analysis of the memory access behavior of several multi-media and robotic embedded-systems applications.**

I. Introduction

Task migration, the process of transferring a running task from one core to another, has been thoroughly researched in the field of distributed computing [3], [17], [18] and is now re-gaining crucial importance in the domain of multi core architectures. These architectures need to adjust their behavior based on runtime observations to account for scenarios that are hard to predict at design time [9], which is commonly referred to as *runtime adaptivity*. Examples include runtime adaptivity to achieve load balancing, to facilitate dynamic thermal management or to resolve network congestion problems in network-on-chip architectures. Such runtime adaptive systems require frequent (3-10 migrations per second in [8]) task migration between the cores [8], [11].

Supported by Intel's projection that the number of cores per chip will double every two years [2], recent trends indicate that future multi core architectures will consist of hundreds or even thousands of cores on the same chip [6]. Such systems may have a distributed memory architecture where each core has a private memory, supported by the increasing on-chip memory capacity of state-of-the-art systems, e.g., Intel's Core i7 Gulftown integrates 13 MiB of on-chip memory. A further increase in the chip transistor count following Moore's law will lead to large on-chip memory capacities in future systems, while a 3D stacking of memories may further increase this capacity [13]. Systems depart from the paradigm of completely sharing memory resources because this does not scale well with a large number of cores. It forms an access bottleneck that can significantly impair the performance of a system [12]. In a distributed memory architecture, task migration requires the transfer of the *task context*, the memory contents that describe the state of a task. Hybrid memory systems with both distributed on-chip memory and multiple off-chip memories may need to transfer memory contents between off-chip memories if a task is migrated between cores that do not share the same off-chip memory. Thus, task migration comes at the cost of a performance penalty and increased on-chip communication. The policy *how* to transfer the task context is of crucial importance to the performance of the system. Frequent task migration may contribute significantly to the network traffic which can increase the power consumption and the temperature of the system. Thus, a task migration mechanism needs to allow the middleware of a runtime adaptive system to balance the performance versus the increased bandwidth requirements adaptively based on runtime observations. These include chip temperature, battery capacity, or the load of the communication fabric. This paper focuses on large multi core architectures that use a network-on-chip [9] and employ distributed memories that are private to each core. Migration of tasks between heterogeneous cores (different ISA, address width, bit/byte endianess, etc.) requires different program codes and a modified task context. State-of-the-art systems solve this through checkpointing [7], [14], [16] and application-level save-restore mechanisms [7], while there exists no mechanism that is transparent to the application. This paper, however, focuses on reducing the latency and on controlling the trade-off with the bandwidth requirements and the delay that arise from task migration between homogeneous cores and does not tackle issues that arise from heterogeneity. This is motivated by the observation that the intrusive application-level migration mechanism presented by [7] requires comparably long latencies (more than 100-1000ms) and thus, frequent migrations are not feasible. Therefore, this paper presents a middleware-level migration mechanism that can be combined with application-level migration between heterogeous cores to achieve fast, transparent migration between identical cores and a slower, application-level migration between heterogeneous cores.
In the following sections, we will discuss the motivation for runtime adaptive task migration and discuss related work. Section IV presents the details and algorithm used in CARAT guided by a memory access behavior study, followed by experimental results, with their interpretation concluding this paper in Section VI.

II. Motivation and Novel Contribution

Motivations for task migration include:
Load Balancing across cores can increase the performance, re-

†Part of this work was funded by the German Science Foundation (DFG), SFB 588 (Humanoid Robots).

duce the peak temperature or guide the aging of a system [11]. Thermal considerations can be especially important [15] as cores may heat up each other, leading to increased leakage power, transient errors or permanent physical damage.
Reliability considerations driven by system health indicators, runtime error checking, and on-chip communication fault statistics may dictate to avoid the excessive use of some cores. In order to gracefully handle such situations, tasks may need to be migrated to other parts of the chip.
Adaptivity at runtime requires task migration to allow an adaptive response to changing system properties, e.g. the link utilization or vitality in systems with on-chip networks, the availability of working memory, hardware accelerators, etc.
Variability in sub-22nm CMOS causes a decreased predictability of the exact functional, electrical and thermal behavior of a system in the pre-silicon phase [6]. Runtime adaptivity with task migration can address these issues.
Power consumption can be reduced when a multi core system moves tasks away from cores (even though the load might be perfectly balanced) to allow their shut-down.

For systems that require frequent task migrations, their latency has significant impact on the overall performance of the system and thus, the latency must be as small as possible. In a distributed memory architecture, the total latency of a task migration can be described as

$$L = D \times (L_P + L_S + L_R + (N_{PF} \times L_{PF}))$$

where, D is the *Manhattan distance* of the cores. L_P denotes the latency caused by the transfer of the program code. Program code typically does not change during runtime and can be transferred while the task is running. Thus, L_P can be assumed to be 0. L_S and L_R are the latencies caused by sending the stack (S) and registers (R), which need to be transferred while the task is paused. The performance of a task migration mechanism is dominated by the transfer of the heap [5]. ($N_{PF} \times L_{PF}$) denotes the number of heap *page faults* multiplied with the latency of a single *page fault*, assuming memory pages are transferred from source to destination in parallel to the execution. A task migration mechanism can reduce the number of *page faults* by optimizing the policy for sending the required memory pages. The energy consumption of a multi core system using a network-on-chip communication paradigm is to about 30% driven by the communication volume [19]. Thus, increased bandwidth requirements not only slow down the communication, can cause network congestion and transmission failures, but also largely increase the energy consumption of the system. Bandwidth requirements and the migration latency are negatively correlated as transferring additional pages decreases the probability for a *page fault* and vice versa. This trade-off needs to be controllable in a runtime adaptive manner, e.g. a system can choose to decrease the energy consumption and to accept a higher migration latency in order to allow the prolonged operation on a low battery capacity, to reduce the temperature, etc.

Our **novel contributions** of this paper are as follows:

1) We present a context-aware runtime adaptive task migration mechanism (CARAT) that allows to largely reduce the migration latency.
2) This mechanism incorporates an approach to control the task migration delay and the tradeoff between the latency and the task migration bandwidth requirements in a runtime-adaptive manner.
3) The transfer policies of CARAT are based on a memory access behavior analysis of the state-of-the-art multimedia, compression, and embedded applications.

III. Related Work

Related work in [8] concentrates on choosing targets and deriving decisions when to trigger task migration and [11] discusses distributed algorithms to migrate tasks in multi core architectures to balance the temperature of the chip. Runtime adaptivity for computer systems has been introduced by [20] and has been used to for application mapping in [10]. Runtime adaptive on-chip routing is researched and implemented in [9]. State-of-the-art task migration mechanisms like [1], [5] do not include runtime adaptivity and propose software checkpointing. Here, the developer explicitly specifies the state information to be transferred and how it is represented. Other approaches require message-based producer/consumer programming models and tasks empty their work queue before they are migrated [16]. [7] requires an application-level context save-restore mechanism. The drawbacks of these approaches include (a) an increased development complexity and the use of system-specific migration libraries. They (b) also do not incorporate runtime adaptivity and thus cannot account for the current system state. Additionally, the enforcement of programming models (c) makes a port of existing applications to multi core architectures much more complicated. The message- and checkpoint-based approaches suffer (d) from an unpredictable and potentially very large task migration delay that might render frequent task migration impossible. (e) The performance of these approaches depends on the individual implementations and thus, it is unpredictable on system level.

A middleware-level task migration framework that works similarly to task switching in preemptive multithreading systems can allow a wider variety of applications, simplify application development, and deliver a task migration delay more predictable on system-level. Transparent task migration mechanisms have been researched in the domain of distributed systems [3], [4]. In [3], source and destination cores negotiate for the required resources and, if successful, transfer the task's address space to the destination. They use a complex marshaller and implement an *establishment* phase to translate pointers. These systems implement the following migration mechanisms: The *eager-copy* mechanism pauses the task and sends the entire program context to the destination core, where the task is resumed once this has completed. A bandwidth-reduction *lazy-copy* mechanism initially only transfers the minimum task context (program code, stack and registers) and sends missing pages when accessed on the destination. A latency-optimized *pre-copy* mechanism transfers the entire program context while the task runs on the source and re-sends pages modified after their transfer while the task is paused. [17] proposes task migration with significantly reduced overhead: It introduces a *post-copy* mechanism that transfers only

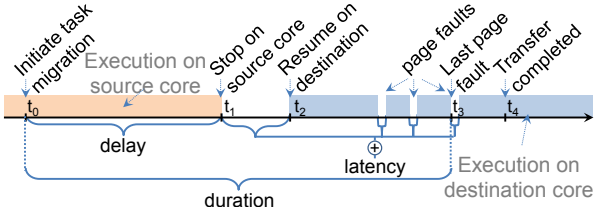
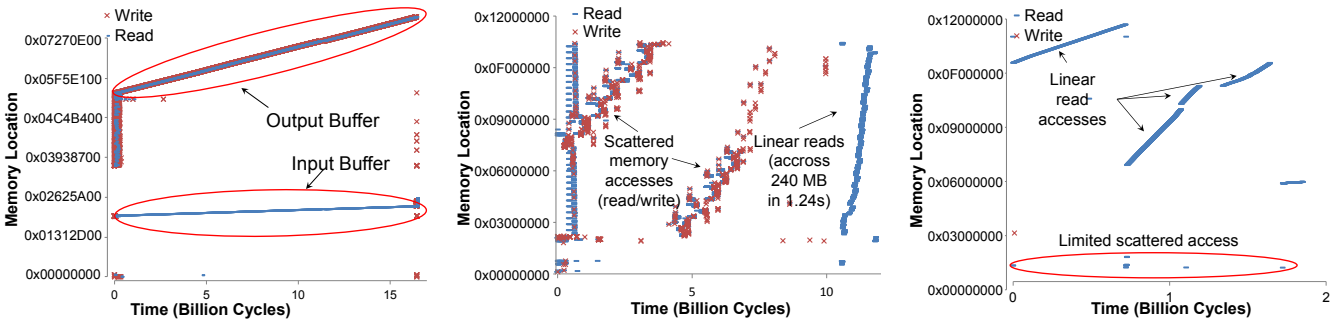


Fig. 1. Latency, duration and delay of task migration



(a) 7Zip encoder, 85 MB binary file (b) x264 encoder, 96 MB YUV movie (CIF res.) (c) Robotic application, 640x480 stereo vision

Fig. 2. Memory access behavior after randomly-chosen task migration initiations

part of the task context while the task's execution is paused, and sends other data once the task has been resumed at the destination core. The *post-copy* mechanism uses a *page fault* mechanism where, whenever a page is accessed that has not yet been transferred, the operating system *initiates* its transfer. Task migration in multi core architectures, however, has fundamentally different requirements than in distributed systems. Here, the migration latency is of crucial importance as it may be invoked frequently. Aspects of data consistency and security have only marginal impact as the data is not transferred over unreliable and untrustworthy wide-area networks. To account for these different requirements, to reduce the latency of task migration, and to control the maximum migration delay and the latency-bandwidth tradeoff adaptively, we are proposing CARAT. It adapts the page migration concepts from [17], [18] by incorporating a thorough memory access behavior analysis for the state-of-the-art applications.

IV. The CARAT Mechanism

CARAT allows frequent migrations with a low impact on system performance by providing a runtime adaptive mechanism to largely reduce the migration latency. This latency is mainly driven by the waiting time for pages accessed on the destination but not yet transferred (*page fault*) because the size of the heap memory is typically larger by the order of magnitudes than the registers and the stack. Thus, achieving a low migration latency can only be achieved by intelligently choosing the order in which to transfer the memory pages. It should match the order of the accesses on the destination core as closely as possible. Runtime adaptivity is achieved by introducing a latency/bandwidth tradeoff parameter $\alpha \in [0, 1] \subseteq \mathbb{R}$. A system should use a high α value if the performance of the task migration is of major concern. It may, however, reduce α to conserve energy by reducing the amount of memory pages that are transferred pro-actively, which means prior to causing a *page fault*. A value of $\alpha = 0$ transfers only pages when they are accessed, and thus CARAT converges to the *lazy-copy* mechanism in this case. To measure the performance of CARAT and to compare it to existing mechanisms, we analyze the performance penalty that it causes to the application (**latency**), the total time spent for the task migration from the initiation to the last page fault or completion of the transfer, whichever comes first (**duration**), the time from the initiation of the task migration to the stopping of the task on the source core (**delay**), and the amount of data it transfers (**bandwidth**). We ignore the time required to transfer stack and registers. The latency of a task migration is the time where the execution of the task is disrupted for task migration reasons. The duration of a task migration is the time span between the initiation and the completion of a task migration. The delay is the time span between the initiation of the migration to the stopping of the task on the source core. This is a crucial measure for emergency task migration mechanisms initiated to resolve high peak temperatures on a core, for example. The bandwidth a task migration requires is the total traffic (in MiB) that arises due to transferring the task from source to destination. The relation of latency, duration, and delay is depicted in Figure 1.

A. Memory Access Behavior Analysis

The memory access behavior of an application is critical to the performance of the different task migration mechanisms, both in terms of latency and in terms of communication overhead. This is because a disparity in the page transfer policy and the access behavior increases the probability of *page faults*, while the transfer of pages that are no longer being used by the application unnecessarily increases the bandwidth requirements. Therefore, we have analyzed the memory access behavior of different applications with different input datasets.

We have chosen a complex, state-of-the-art multi-media application (an x264 encoder), the 7Zip LZMA implementation and a state-of-the-art embedded systems robotic application that performs pipelined computer vision, feature extraction and stereo matching. Figure 2 shows the memory accesses of these applications after a randomly triggered task migration. The figures only show accesses to blocks (a section of memory allocated at once) allocated prior to the task migration as other blocks are allocated on the destination core.

A first observation shows that different applications have a very different memory access behavior, ranging from a linear behavior of the 7Zip application to a very random, irregular memory access of the x264 video encoder.

From these observations, we gain the following insights: (a) trivially, not all pages are required at the same time, and (b) when a page is accessed, it is *likely* that succeeding pages will be accessed soon. (c) we find that an application might separate read- and write buffers and thus, buffers that have predominantly been written to are unlikely to be accessed with heavy read accesses in the near future. (d) some applications use small buffers to store frequently used operands, such as filter kernels, that may be accessed excessively. (e) We also find that memory blocks are mostly accessed from front to back, not in reverse order. (f) We recognize that access

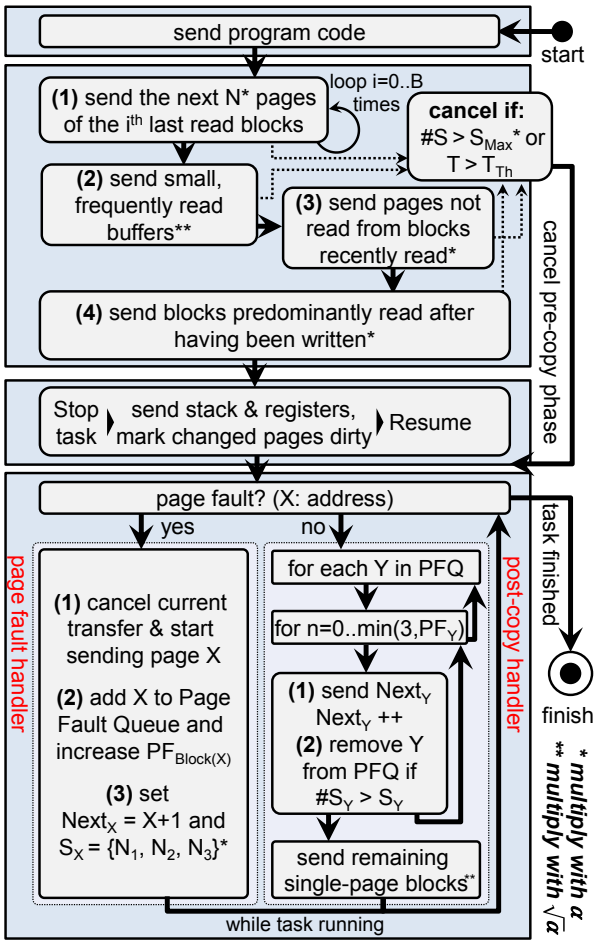


Fig. 3. Schematic view of the CARAT algorithm

behavior differs largely across applications and thus, a task migration mechanism should not suffer from unexpected accesses beyond the performance of the existing mechanisms and should not be tied to a specific application if a broad applicability is desired.

B. Algorithmic Solution

The CARAT mechanism rationale is detailed in pseudo-code in Algorithm 1 while the algorithmic flow is depicted graphically in Figure 3: First, until the maximum task migration delay is reached (delay threshold) or until a maximum amount of pages is transferred (bandwidth threshold), the mechanism sends pages in parallel to the execution of the task similar to the *pre-copy* mechanism. In contrast to this mechanism, however, the pages are chosen based on the following rationale: Until the specified maximum migration delay is reached, do the following: First, send the next $N \times \alpha$ pages of the B most recently read blocks. Second, send small buffers that have been read frequently, where the total number of buffers to transfer is multiplied by $\sqrt{\alpha}$. Third, send α (meaning $\alpha \times 100\%$) of the remaining pages that have not yet been accessed for read access in blocks where other pages have recently been read. Fourth, send α of the blocks that have been predominantly read after having been written. For the design parameters N and B, we have chosen the values 8 and 4. Optimal parameter values needs to be derived through design-space exploration, which is beyond the scope of this paper. Then, the task is paused on the source core, the stack

Algorithm 1 CARAT transfer algorithm

Parameters:

α	Latency/bandwidth tradeoff parameter
S_{Max}	Maximum number of pages in pre-copy phase
Th_{Delay}	Maximum delay threshold

Definitions:

T	Current time / cycle counter
$\#S$	Number of transferred pages
S_B	Pages to transfer from block B
$Block(X)$	Block containing page X
Γ_B	Largest page address of block B
PF_B	Page fault counter for block B
PFQ	Page-fault queue
$Next(X)$	Next page to send for page fault at X
N_1, N_2, N_3	Design-time parameters (page-buffer size)

Phase 0: *Initialization*
Transfer program code
Phase 1: *Adaptive pre-copy phase*
while $T < Th_{Delay}\ AND\ \#S < S_{Max}$ **do**
  $X \leftarrow$Choose next page$_\alpha$
  Transfer X, break if $X = \emptyset$
  $\#S \leftarrow \#S + 1$
end while
Phase 2: *Switch execution*
Pause task
Transfer register contents and program stack
Mark transferred but overwritten pages as missing[a]
Continue task execution on destination core
Phase 3: *Post-copy / page fault handler phase*
while Task running **do**
  if Page fault occurred at X **then**
    `// Page fault handler`
    Cancel current transfer
    $PFQ \leftarrow X \cup PFQ$
    $PF_{Block(X)} \leftarrow PF_{Block(X)} + 1$
    $Next(X) \leftarrow X$
    if $PF_{Block(X)} = \{1, 2, 3, > 3\}$ **then**
      $S_X \leftarrow \{X + N_1, X + N_2, X + N_3, \Gamma_{Block(X)}\}$
    end if
    `//Scale upper transfer boundary with` α
    $S_X \leftarrow MAX(1,\ S_X \times \alpha)$
  end if
  for all $Y \in PFQ$ **do**
    `// Serve page fault queue (1-3 pages)`
      `Interrupt if another page fault occurs`
    for $n = 0$ to $MIN(3,\ PF_{Block(Y)})$ **do**
      if $Next(Y) > S_Y$ **then**
        $PFQ \leftarrow PFQ \cap Y$
        Exit For Loop
      else
        Transfer $Next(Y)$
        $Next(Y) \leftarrow Next(Y) + 1$
      end if
    end for
  end for
  for all $\alpha\times$ Single-page blocks not yet sent **do**
    Send next single-page block α
  end for
end while

[a] Hardware support in the MMU is required.

and register contents are transferred and then the execution is resumed on the destination core.

The unified, adaptive *post-copy* and *page fault* handler transfers selected pages in parallel to the resumed execution and responds to page faults. As the memory access behavior analysis shows that a fair fraction of allocated memory might not be used anymore (other than being freed) after the task migration, the adapted *post-copy* send handler does not send the entire memory of a task. This allows to reduce the total amount of bandwidth requirements. The unified *post-copy* / *page fault* handler transfers as follows: When a *page fault* occurs, the current transfer is canceled and the missing page as well as some succeeding pages are sent immediately, while the number of succeeding pages to be sent depends on the design parameters N_1, N_2, and N_3: For the first *page fault* for this block, the missing page and the succeeding $N_1 \times \alpha$ pages are

sent. For the second *page fault*, the succeeding $N_2 \times \alpha$ pages are sent and for the third fault, the succeeding $N_3 \times \alpha$ pages are sent. From the fourth *page fault* in a block, the entire block is transferred. When this priority-sending of missing pages is not yet completed before a *page fault* in a different block occurs, each priority request is added to a queue which is served in a round-robin fashion, weighted with the number of *page faults* in each block to time-multiplex the communication with a per-page granularity. When the priority-sending has completed, single-page blocks are transferred. This accounts for the observation that applications may use a fair number of small blocks, which could increase the *page fault* count tremendously as the block-wise sending described in the *page fault* handler would fail to deliver a good performance.

C. Runtime Adaptivity

Runtime adaptivity is achieved through the tradeoff parameter $\alpha \in [0, 1]$ that balances latency and bandwidth requirements. For $\alpha = 1$, CARAT reduces the latency as much as possible, while smaller values steadily increase the latency but decrease the bandwidth requirements. It converges to the *lazy-copy* mechanism for $\alpha = 0$ as no pages are transferred without being requested through a *page fault*. Note that CARAT exposes this runtime adaptivity to the operating system and itself does not choose α based on runtime observations.

To achieve this behavior, each page transfer that is not a response to a *page fault* is scaled (multiplied) by α, while the small buffer advance transfers in the adapted *pre-copy* phase are scaled with $\sqrt{\alpha}$ because small buffer transfers cause relatively low communication traffic while delivering a high probability of reducing *page faults*. However, $\alpha = 1$ does *not* imply excessive bandwidth requirements; they are typically smaller than those of the *pre-copy* and *post-copy* mechanisms while delivering a significantly reduced latency. Thus, $\alpha = 1$ should be considered the default and lower α values should be used to conserve energy, to reduce the heat of the system or to account for high network loads. The runtime adaptivity that CARAT provides through α, the maximum *pre-copy* transfer volume and delay threshold are instrumented by the operating system. While the discussion how to derive the parameter values from system state observations is beyond the scope of this paper, a relationship between the battery level and α seems natural. This flexibility allows runtime adaptive systems to control their power consumption, performance and temperature tradeoff at a finer grain. CARAT is, however, limited to multi core architectures with distributed memories.

V. Experimental Results

We have analyzed and measured the behavior of CARAT and compared it to implementations of the *post-copy*, *pre-copy*, and *lazy-copy* mechanisms for the x264 encoder, the 7Zip compression encoder and for the embedded-systems robotic application. We have chosen the single-threaded implementations of these programs as we analyze the migration of a single task. More precisely, we analyzed the memory access behavior after a number of randomly-chosen task migration initiations. We analyzed 12 task migrations per application to get a reasonably large sample size.

Our experimental target architecture features 1 GHz processing cores and a 4 GBit/s link bandwidth. Source and destination cores are directly connected (single-hop distance, this is not a constraint) and the network serves a 50% network load.

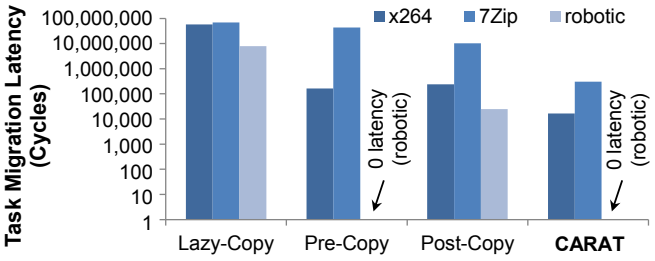


Fig. 4. Latency of the different task migration mechanisms

For the experiments, we have not constrained the maximum task migration duration and set $\alpha = 1$.

A. Monitoring Environment

We have collected the memory access behavior statistics on a shared-memory system as the application behavior does not depend on the target architecture. To achieve a lightweight and non-intrusive memory access statistics, x86 debugging capabilities are utilized. More precisely, guard pages are used that trigger an interrupt when accessed for the first time. Windows automatically removes the guard page flag after the iterrupt has been handled. The interrupt handler merely logs the address of the accessed page and a copy of the processor's cycle count register (RDTSC) to a pre-allocated memory structure. To get an accurate overhead estimate, we have compared the total runtime of 100 runs of the simulation framework with enabled and disabled memory access monitoring. As a result, the memory access behavior monitoring requires approximately 25.000 cycles per first-time page access, while a 42-second run of the x264 encoder triggered 27667 interrupts, causing a total application slowdown of roughly 0.5%. Note that memory block sizes are known to the middleware as the OS keeps track of the application's allocations.

Table I shows the results for the analyzed dimensions of latency, duration, delay and bandwidth of CARAT compared to the *lazy-copy*, *pre-copy*, and *post-copy* mechanisms. These numbers only include heap memory transfer as stack & register transfer times are constant and merely offset these numbers. The best result in each category is written green*, while the worst result is written in red color$_!$. During our experiments, we have found the values 8, 32 and 128 for the parameters N_1, N_2, and N_3, respectively, to deliver good results.

Interpreting the results depicted in Table I, we see that

		Results			
		Lazy-Copy	Pre-Copy	Post-Copy	**CARAT**
x264	**Page Faults**	14038$_!$	0*	2	2
	Pages Sent	14038*	26116$_!$	26097	24923
	Latency^a	5,76E+07$_!$	164120	238380	16398*
	Duration^a	1,34E+11$_!$	2,14E+08*	1,34E+11$_!$	1,33E+11
	Delay^a	0*	2,14E+08$_!$	0*	2,04E+08
	Bandwidth^b	54,84*	102,02$_!$	101,94	97,36
7Zip	**Page Faults**	16960$_!$	0*	1240	37
	Pages Sent	16960*	52734$_!$	47394	47389
	Latency^a	6,96E+07$_!$	4,38E+07	1,02E+07	303118*
	Duration^a	2,22E+11$_!$	4,32E+08*	2,22E+11	2,22E+11
	Delay^a	0*	3,88E+08$_!$	0*	7,90E+07
	Bandwidth^b	56,25*	205,99$_!$	185,13	185,11
robotic	**Page Faults**	1917$_!$	0*	3	0*
	Pages Sent	1917*	4085$_!$	4085$_!$	4072
	Latency^a	7,87E+06$_!$	0*	24660	0*
	Duration^a	1,85E+09$_!$	3,35E+07*	1,85E+09$_!$	1,72E+09
	Delay^a	0*	3,35E+07$_!$	0*	3,34E+07
	Bandwidth^b	7,49*	15,96$_!$	15,96$_!$	15,91

^{a} Units: clock cycles ^{b} Units: MiB

TABLE I
Results for the different task migration mechanisms

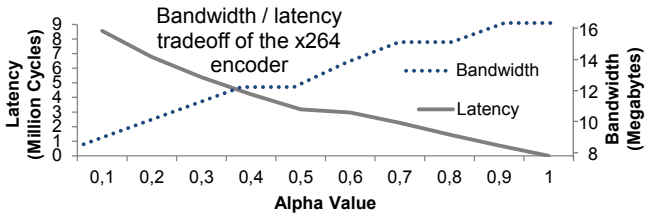


Fig. 5. Impact of α on the latency / bandwidth tradeoff

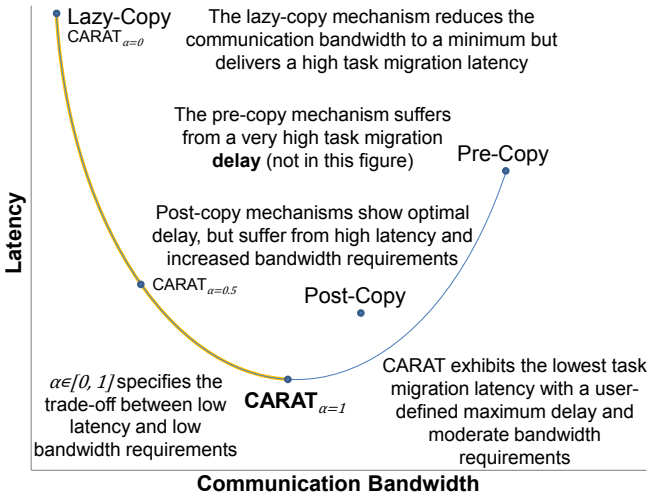


Fig. 6. Latency and bandwidth tradeoff

CARAT easily achieves the lowest task migration latency, which is also depicted in Figure 4, and, compared to the *lazy-* and *post-copy* mechanisms, the least amount of *page faults*. This is achieved by subtly selecting the pages that are subject to pre- and parallel transfer. On an average, this reduces the latency by 93.12%, 97.03% and 100% for the x264 encoder, the 7Zip encoder and the robotic application, respectively, when compared to the *post-copy* mechanism, and 90.01%, 97.03% and 0%, compared to the best state-of-the-art mechanism in each dimension. We can also see that CARAT never delivers the worst result in each dimension, which shows that the goal not to suffer from a penalty beyond the other mechanisms is met. Additionally, as a maximum duration of the task migration can be explicitly specified, CARAT is especially applicable for runtime adaptive systems and allows to keep real-time constraints more easily.

B. Runtime Adaptivity Results

The effects of scaling the α value to tradeoff between the task migration latency and the bandwidth requirements of the algorithm is depicted in Figure 5. The α scaling provides a monotone relationship to the bandwidth reduction and the latency increases, which are inversely related. This property also allows to estimate the impact of changing the α value. The bandwidth requirements for $\alpha = 0$ match the bandwidth requirements of the *lazy-copy* mechanism, while $\alpha = 1$ requires less bandwidth than the *pre-copy* and *post-copy* mechanisms with significantly reduced task migration latency. Figure 6 illustrates the tradeoff between the latency and the associated bandwidth. However, this figure does not show the large delay penalty that the *pre-copy* mechanism comes with.

VI. Conclusions

This paper presents CARAT, a novel task migration mechanism that allows to largely reduce the migration latency and to control the delay and the latency/bandwidth tradeoff. This allows multi core systems to instrument task migration to achieve dynamic thermal management, load balancing or for reliability enhancement. CARAT allows the operating system to account for the current system state adaptively at runtime. It does not support to migrate tasks between heterogeneous cores, but is orthogonal to application-level mechanisms that tackle this issue. Our experiments show that compared to existing work, CARAT delivers the lowest task migration latency while it never demands the longest duration, delay or highest bandwidth requirements. Furthermore, the combination of a *pre-copy* and a unified *post-copy* and *page fault* handler allows to control the maximum delay. Consequently, task migration may be used in a runtime-adaptive manner in multi core architectures with a largely reduced latency.

References

[1] A. Acquaviva et al. Assessing Task Migration Impact on Embedded Soft Real-Time Streaming Multimedia Applications. *EURASIP Journal on Embedded Systems*, 2008.

[2] A. Agarwal. CPU Designers Debate Multi-Core Future. *EE Times*, 2008.

[3] Y. Artsy and R. Finkel. Designing a Process Migration Facility: The Charlotte Experience. *Computer*, 22(9), 1989.

[4] A. Barak. Scalable Cluster Computing with MOSIX, for Linux. In *In Proceedings of Linux Expo 99*, 1999.

[5] S. Bertozzi et al. Supporting task migration in multi-processor systems-on-chip: A feasibility study. In *Proceedings of the Conference on Design, Automation and Test in Europe*, 2006.

[6] S. Borkar. Design perspectives on 22nm CMOS and beyond. In *Proceedings of the 46th Design Automation Conference*, 2009.

[7] P. Bungale et al. An Approach to Heterogeneous Process State Capture/Recovery to Achieve Minimum Performance Overhead During Normal Execution. In *Proceedings of Parallel and Distributed Processing Symposium*, 2003.

[8] D. Cuesta et al. Adaptive Task Migration Policies for Thermal control in MPSoCs,. In *Proceedings of the IEEE 2010 Annual Symposium on VLSI*, 2010.

[9] M. A. A. Faruque, T. Ebi, and J. Henkel. Run-Time Adaptive On-Chip Communication Scheme. In *Proceedings of the 2007 IEEE/ACM International Conference on Computer-Aided Design*, 2007.

[10] M. A. A. Faruque, R. Krist, and J. Henkel. ADAM: Run-Time Agent-Based Distributed Application Mapping for On-Chip Communication. In *Proceedings of the 45th Design Automation Conference*, 2008.

[11] Y. Ge, P. Malani, and Q. Qiu. Distributed Task Migration for Thermal Management in Many-Core Systems . In *Proceedings of the 47th Design Automation Conference*, 2010.

[12] J. Howard et al. A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. In *Proceedings of the 57th International Solid State Circuits Conference*, 2010.

[13] G. H. Loh. 3D-Stacked Memory Architectures for Multi-Core Processors. In *Proceedings of the 2008 International Symposium on Computer Architecture*, 2008.

[14] J.-Y. Mignolet et al. Infrastructure for Design and Management of Relocatable Tasks. In *Proceedings of the Conference on Design, Automation and Test in Europe*, 2003.

[15] F. Mulas et al. Thermal Balancing Policy for Multiprocessor Stream Computing Platforms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 28(12), 2009.

[16] V. Nollet et al. Centralized Run-Time Resource Management in a Network-on-Chip Containing Reconfigurable Hardware Tiles. In *Proceedings of the Conference on Design, Automation and Test in Europe*, 2005.

[17] M. Richmond and M. Hitchens. A new process migration algorithm. *SIGOPS Operating Systems Review*, 31(1), 1997.

[18] J. M. Smith. A Survey of Process Migration Mechanisms. *SIGOPS Operating Systems Review*, 22(3), 1988.

[19] V. Soteriou and L.-S. Peh. Design-Space Exploration of Power-Aware On/Off Interconnection Networks. In *Proceedings of the IEEE International Conference on Computer Design*, 2004.

[20] S. R. White et al. Autonomic Computing: Architectural Approach and Prototype. *Integrated Computer-Aided Engineering*, 13(2):173–188, 2006.