Work in Progress: Malleable Software Pipelines for Efficient Many-core System Utilization

Janmartin Jahn, Sebastian Kobbe, Santiago Pagani, Jian-Jia Chen, Jörg Henkel

To cite this version:

Janmartin Jahn, Sebastian Kobbe, Santiago Pagani, Jian-Jia Chen, Jörg Henkel. Work in Progress: Malleable Software Pipelines for Efficient Many-core System Utilization. The 6th Many-core Applications Research Community (MARC) Symposium, Jul 2012, Toulouse, France. pp.30-33. hal-00719027

HAL Id: hal-00719027
https://hal.science/hal-00719027v1

Submitted on 18 Jul 2012

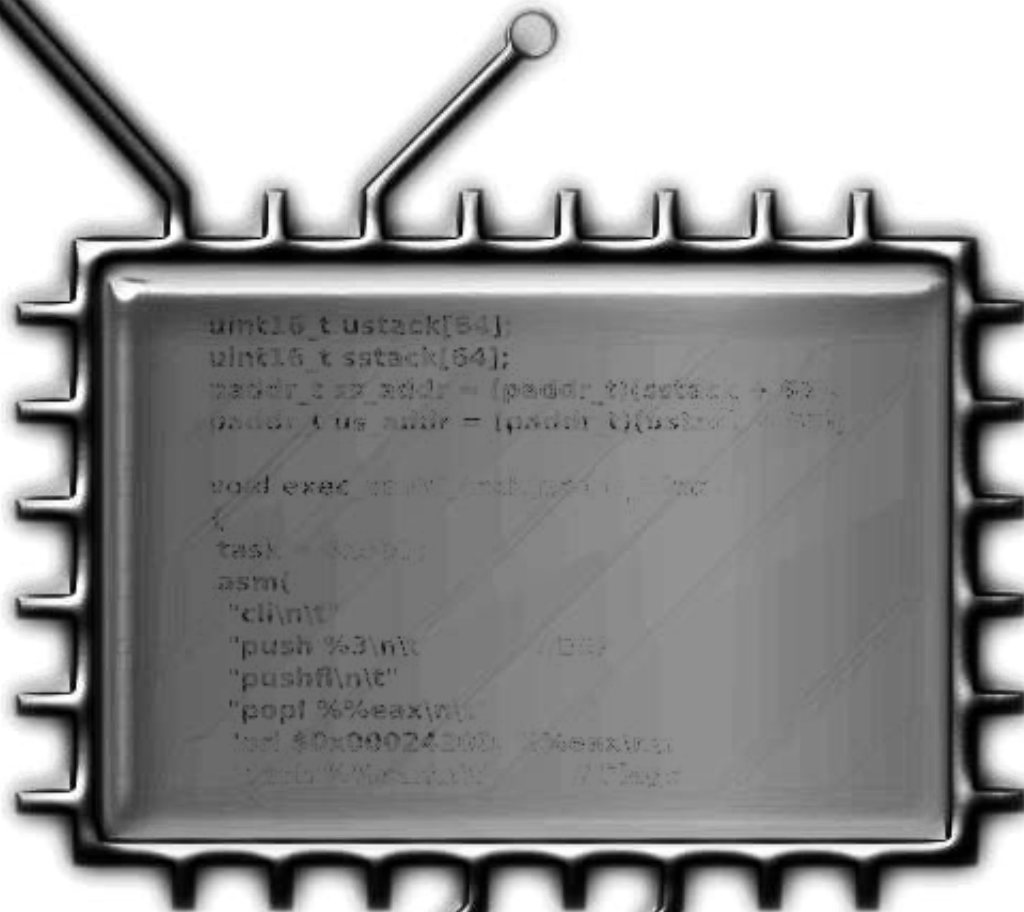
HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Proceedings of the 6th Many-core Applications Research Community (MARC) Symposium

http://sites.onera.fr/scc/marconera2012

July 19th–20th 2012



ISBN
978-2-7257-0016-8

ONERA

THE FRENCH AEROSPACE LAB

 ISBN: 978-2-7257-0016-8

Work in Progress: Malleable Software Pipelines for Efficient Many-core System Utilization

Janmartin Jahn, Sebastian Kobbe, Santiago Pagani, Jian-Jia Chen, Jörg Henkel
Karlsruhe Institute of Technology (KIT), Germany

***Abstract* — This paper details our current research project on the efficient utilization of many-core systems by utilizing applications based on a novel kind of software pipelines. These pipelines form malleable applications that can change their degree of parallelism at runtime. This allows not only for a well-balanced load, but also for an efficient distribution of the cores to the individual, competing applications to maximize the overall system performance. We are convinced that malleable software pipelines will significantly outperform existing mapping and scheduling solutions.**

***Index Terms* — Parallel architectures, multicore processing, pipeline processing, multiprocessing systems.**

I. Introduction and Motivation

Running multiple applications efficiently on a many-core system requires careful decisions about the distribution of the availablecores among the running applications becauseof the following reasons: an inefficient distribution of coresmay lead to an imbalanced load, thus leaving resources idle, or to a reduced system performance (even though the load may be perfectly balanced) when assigning few cores to demanding applications while giving more coresto applications that hardly benefit from them. Thelatter may be due to different application *efficiencies*(i.e. the speed-up per core), which areexpressed bythe corresponding *speed-up functions* that describe how the performance of an application dependson the number of cores assigned to it.Thus, thesedecisions largely impact the overall performanceof many-core systems. Parallel applications often are not able to achieve a linear speed-up with the number of cores [1], i.e. the efficiency decreases. Consequently, when running multiple applications, it is crucialto distribute the cores in a way that the overall efficiency is maximized, which is illustrated in Figure 1 for the combined (system) efficiency when running two competing applications.

There are three kinds of parallel applications: a) statically parallelizedapplications that are only able to execute on a fixed number of cores, b) moldable applications whose degree of parallelism can be defined at the start-up time of the application,and c) malleable applications that can change their degree of parallelism (from now on, we will call this *resizing*) at runtime.With varying and/or dynamic workloads, malleable applications allow the highest overall system efficiency as the optimal number of cores could be assigned to each application whenever the system state changes [2].

However, the effort and the costs of resizing have to be considered because they may be manifold and include task migration and workload distribution, which may be especially hurtful for many-core architectures with distributed memories [3]. Depending on the kind of application, resizing is not possible at arbitrary points. Today, malleable applications are often used in high-performance computing and grid-computing environments, often work on large datasets, and run for long periods of time (e.g., hours or days). Here, the absolute time required for the re-distribution of cores is of minor importance as the total system throughput in long periods of time is crucial. However, due to this property, such applicationsare not suitable for highly interactive systems or mobile devices: here, systems need to be responsive, so the time spent for resource re-distribution needs to be minimal and theefficiency of the system needs to be improved quickly (and not in the long term) to satisfy user demands. In typical high-performance computing environments, large re-distribution timesare tolerable whencompared to the total application runtime. Instead, we address systems where applications must beresized at a low overhead to allow frequent variations – e.g., when new applications enter the system,the system state changes, or a user starts interacting with the device.

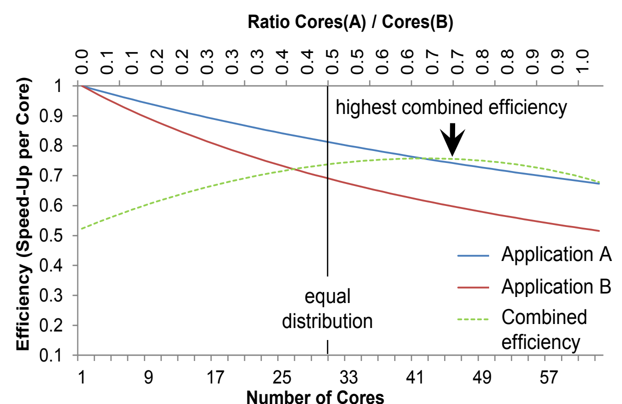


Figure 1 - Combined efficiencies of two competing applications

To make decisions about which application should be allowed to use how many resources, state-of-the-artresource management schemes for malleable applications (such as [4, 5]) need to know how well the application will perform with these resources, i.e. thespeed-up function must be known. This knowledge could be obtained by offline profiling or online

monitoring. After distributing the coresamong the runningapplications, the application itself has to balance the loadamong its assigned cores.

Our novel contributions are malleable software pipelines that addressthe three issues of frequent malleability, unknown speed-up functions, and autonomous load balancing.Thisincludes: a) software pipelines that are malleable through decreasing and increasing their level of parallelism at runtime by *fusing*stages, i.e. combining multiple consecutive stages into one larger stage, or splitting previously fused stages (we call this *fission*), b) an online monitoring framework that decides upon the best option for *fusion* and *fission* operations, and c) a compiler that creates those software pipelines using an architecture specification and application profile with a maximum level of granularity.

II. Related Work

There are multiple ways to create malleable applications, but basically they are based on the same principles. On shared memory architectures, creating malleable applications can be done by (e.g., compiler based [6] or library based [7]) exploitation of thread and even loop-level parallelism. The costs for resizing the application are comparably low (because no data has to be migrated between private memories) and allow frequent resizing of the application at the granularity of several hundred milliseconds. For distributed memory systems, where each core has its own, private memory, these approaches are not suitable because the overhead for resizing applications can be prohibitively high. Thus, approaches like Master-Slave parallelization are used [8]. Here, depending on the amount of available cores, more or less workers are created, i.e. resizing is possible at thread level, but the resizing costs are much higher than in a shared memory system.

Consequently, the time between resizing decisions has to be long enough so the gain from the increased efficiency can exceed the corresponding overhead, which makes them very suitable for large scientific computing environments where applications run for hours or days, but less suitable for interactive systems running on MPSoCs. Another possibility is a single program multiple data (SPMD) application architecture [9], where the data is partitioned depending on the available cores. Adjusting the data partition to changing system states is possible but comes at a high cost, which also only allows for seldom changes in the application size.

Adaptive-MPI (AMPI) [10] provides a multitude of 'virtual cores'as the smallest level of parallel granularity, which are prepared at compile time and are mapped to physical coresat runtime. Typically, multiple virtual cores are mapped on one real core. AMPI is based on the Charm++ language [11] and utilizes the same runtime system for load balancing as applications written in Charm++. Resizing of applications implemented with AMPI is transparent to the application itself as only the mapping of virtual to real cores has to be changed and, if necessary, the working set has to be transferred. Therefore, AMPI cannot take advantage of application-specific knowledge (such as choosing optimal migration points between iterations of pipeline stages). AMPI is designed for large, distributed computing environments (e.g., datacenters) where the individual nodes have large main memories and fast cores. In contrast to this, we target architectures with comparably slow individual cores where the access to (off-chip) main memory is very costly and must be reduced to a minimum.

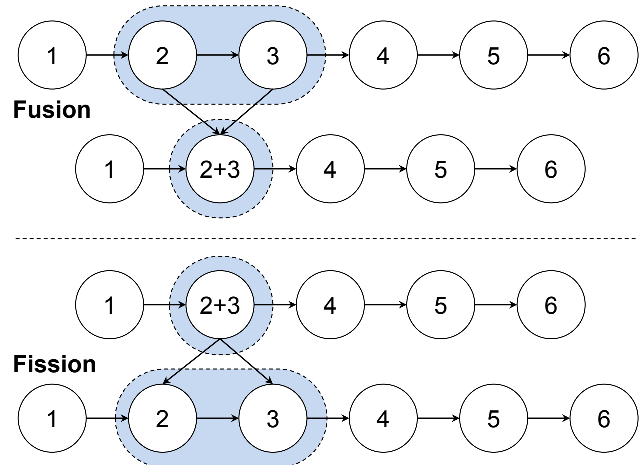


Figure 2 – Basic operations of malleable pipelines

III. Our Approach

Software pipelines present a widely used programming model which is especially suitable to parallelize sequential complex applications for many-core systems with distributed memories that may be private to each core. Multiple compilers, tools and frameworks exist that extract software pipelines from existing applications [12-14]. The presented malleable software pipelines are software pipelines with the following properties:

They support the basic operations of *fusion* and *fission*, as illustrated in Figure 2. A fusion of pipeline stages reduces the degree of parallelism, thus reducing the number of stages by combining two consecutive stages into one. Contrarily, fission increases the degree of parallelism by splitting fused stages.Pipelines are created with a compile-time chosen finest level of granularity, from which no further fissions are possible. Stages can be fused until only one stage remains. In this case, the pipeline is equivalent to the sequential execution of the same algorithm.Malleable software pipelines use runtime monitoring to decide which stages to fuse or which stage to split.

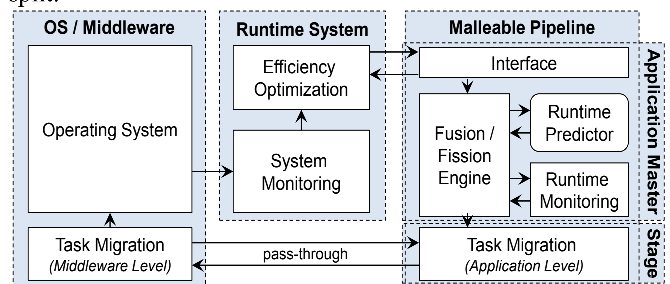


Figure 3 - Block diagram of system components

The system components are illustrated in Figure 3: The runtime system constantly monitors the system utilization and application efficiencies. An efficiency optimization component periodically re-evaluates the current system state and may decide to change the distribution of the system's cores to the applications. Each application consists of one so-called *Application Master* and one process per pipeline stage. These processes are based on the same executable file and may run one or more consecutive pipeline stages. When running multiple stages on one core, no inter-core data communication for these stages is necessary.

The *Application Master* includes an interface for the communication with the runtime system, a *Fusion / Fission Engine*, and a lightweight runtime predictor that predicts the potential performance increase of a fission operation and the potential performance decrease of a fusion operation, and an application level task migration component.

The *Interface* is responsible for the communication with the run-time system and provides it with information such as the maximum level of parallelism (static) and the gradient of the speed-up function (dynamic).

The *Fusion / Fission Engine*is responsible for conducting the fusions and fissions requested by the runtime system. The runtime system specifies which cores may be used or will be taken away from the application, while the Fusion / Fission Engine is responsible for identifying the stages to be fused or split.This decision is based on Runtime Monitoring of the computational and communication demands (from now on, we will call both computational and communication demands 'resource demands') of each individual stage. Using a lightweight Runtime Predictor, the Application Master decides which fusion will result in the least performance degradation and which fission will result in the highest performance increase. To avoid bloated communication volumes and races for communication hardware (such as the Message Passing Buffer in Intel's Single-Chip Cloud Computer [15]), only consecutive stages are fused. When the number of cores assigned to an application changes, all pipeline stages of that application are recombined.

The migration of the pipeline stages is very lightweight as they are migrated after completing an iteration, as software pipelines typically have a peak memory consumption while performing one iteration, and carry little state between iterations. As the executable file is the same for each stage and may dynamically switch from executing one stage to executing several stages, only the state information of the corresponding stages that is carried between iterations needs to be transferred. Consequently, the executable file needs to be transferred only when new cores are made available for an application and the stage fusions may be performed at runtime with very little overhead. In the presented case-study (see Section V), some stages were completely stateless (e.g., image enhancement or transformations which operate on each frame of the video stream individually) while others carried little state (e.g., information about identified objects in the previously processed image) in the worst-case of 38 kilobytes. The size of the state that needs to be transferred for each resizing decision depends on the executed application. However, we find that a typical property of software pipelines is that each stage carries comparably little state between iterations..

IV. Runtime System

The runtime system is responsible for optimizing the efficiency of multiple applications running on a many-core system, thus competing for computational resources. This is performed by resizing the competing applications at runtime. To accomplish this, each application provides information about whether it could efficiently make use of more cores or if the already assigned cores are not used efficiently and a fusion of pipeline stages could be performed to free cores for other applications.If all possible applications and system states are known at design time, these decisions can be made at design time. This allows for optimal allocation, scheduling, and mapping of applications to cores. However, i.e. due to user interactions, changing input data, or unpredictable system state, an online resource management is required to efficiently utilize system resources.In [4], a distributed approach for managing malleable applications on many-core systems with hundreds or thousands of cores has been presented. The scalability of the approach is achieved by avoiding the use of global knowledge or broadcast communication. The approach utilizes an application performance model for parallel applications which considers the theoretical speed-up of an application [1] and the relative placement of the cores of the application to make its decisions.

In the following section we show how well the theoretical performance model and the performance of the real implementation of our malleable software pipelines match. By using real measurements of application performance and resizing overhead (which is currently not exactly modeled in the decision making process) we will be able to enhance the previous approach and apply it to a real many-core system running our malleable software pipelines.

V. Case Study Analysis

We analyze a software pipeline that we generated from a complex, real-world robotic application. The presented software pipeline captures stereo images from two pairs of stereo cameras, uses image enhancement and transformation algorithms, calculates a three-dimensional depth map, and then detects and tracks objects. Due to bandwidth constraints, the images can only be captured sequentially, not in parallel. The application tracks a recognized object and moves the robot to follow it. The 18 individual stages of the pipeline are illustrated in Figure 4. The case study focuses on the behavior of that software. To obtain the results, each pipeline stage had been benchmarked individually on the Intel SCC to obtain the necessary information to calculate the optimal fissions and fusions for any given number of cores.

When executing these stages on multiple cores, the speed-up function is largely sub-linear, as shown in Figure 5. A maxi-

mum speed-up of 9.7x can be achieved when running the application on 18 cores. The efficiency drops when increasing the number of cores, while positive "jumps" in efficiency when increasing the number of cores from 9 to 10 or from 11 to 12 are due to the fact that due to the increased number of cores, the fusions can be changed and otherwise heavily loaded cores have to carry much less load. The values represent stable states for different pipeline lengths. Dynamic effects (e.g., cache effects) that occur directly after resizing an application are not presented in this case study as the runtime system is still work-in-progress.

The overhead of stage fusions is limited as only the state that is carried between iterations needs to be transferred. For the applications we have examined, the magnitude of the required data size ranges from a few bytes to few kilobytes (with 38 kilobytes being the worst case), so the overhead of transferring this state is negligible in most state-of-the-art many-core architectures, which often have a high-bandwidth, low-latency inter-core communication fabric. The overhead of stage fission is purely dominated by transferring the executable file.

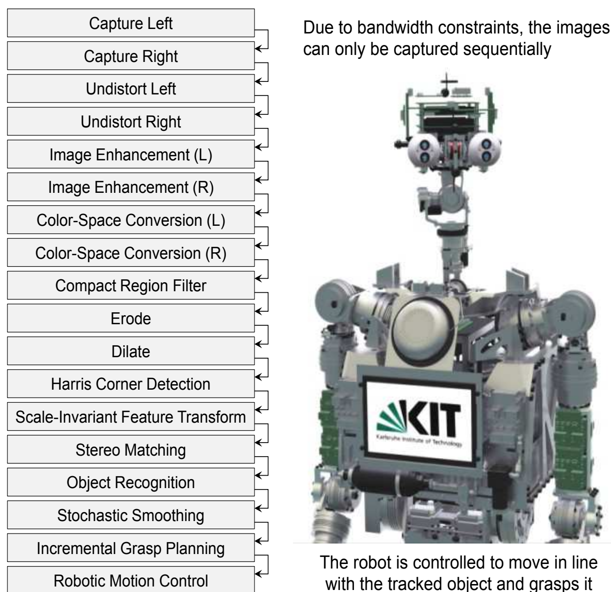


Figure 4 - Pipelined robotic application

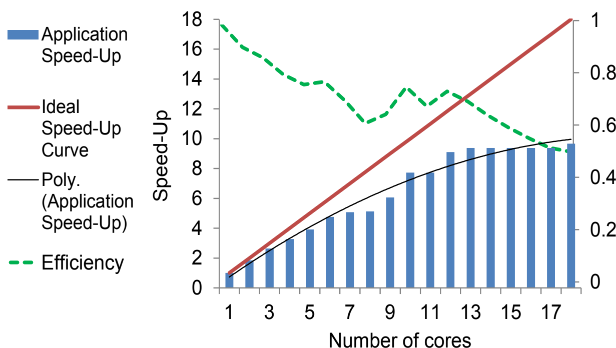


Figure 5 - Speed-up and efficiency of the robotic application

VI. Outlook

The above argumentation urges that the efficiency of the utilization of the computational resources of many-core systems may be greatly enhanced when using malleable applications. In contrast to the state-of-the-art malleable application models, malleable software pipelines are well-suited to MPSoC architectures with comparably slow individual cores, small on-chip memories and highly penalized access to off-chip storage. Shifting malleability to the application layer allows software pipelines to change their degree of parallelism for efficiency optimization without incurring significant overhead because only the state that is carried across multiple iterations needs to be transferred.We are currently implementing the required infrastructure and conduct experiments that compare our malleable software pipelines to the state-of-the-art load balancing and task management systems.

References

[1] A. B. Downey, "A parallel workload model and its implications for processor allocation," in *Sixth IEEE International Symposium on High Performance Distributed Computing*, August 1997, pp. 112–123.
[2] J. Hungershöfer, A. Streit, and J.-M.Wierum, "Efficient resource management for malleable applications",Technical Report, 2001.
[3] S. Borkar, "Thousand core chips: a technology perspective," in *Proceedings of the 44th annual Design Automation Conference (DAC)*, 2007, pp. 746–749.
[4] S. Kobbe, L. Bauer, D. Lohman, W. Schröder-Preikschat, and J. Henkel, "DistRM: Distributed resource management for on-chip many-core systems," in *Proceedings of the IEEE International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Oct. 2011, pp. 119–128.
[5] P. Sanders and J. Speck, "Efficient parallel scheduling of malleable tasks," in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, may 2011, pp. 1156 –1166.
[6] M. W. Hall and M. Martonosi, "Adaptive parallelism in compiler-parallelized code," *Concurrency: Practice and Experience*, vol. 10, no. 14, pp. 1235–1250, 1998.
[7] C. Pheatt, "Intel® threading building blocks," *J. Comput. Sci. Coll.*, vol. 23, no. 4, pp. 298–298, Apr. 2008.
[8] N. Islam, A. Prodromidis, and M. S. Squillante, "Dynamic partitioning in different distributed-memory environments," in *In Job Scheduling Strategies for Parallel Processing*. Springer-Verlag, 1996, pp. 244–270.
[9] L. V. Kalé, S. Kumar, and J. Desouza, "A malleable-job system for timeshared parallel machines," in *Proceedings of the 2nd International Symposium on Cluster Computing and the Grid (CCGrid 2002)*, 2002, pp. 230–237.
[10] C. Huang, O. Lawlor, and L. V. Kalé, "Adaptive MPI," in *In Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03)*, 2003, pp. 306–322.
[11] L. V. Kalé and S. Krishnan, "Charm++: A portable concurrent object oriented system based on c++," in *In Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, 1993, pp. 91–108.
[12] W. Thies, V. Chandrasekhar, and S. Amarasinghe, "A practical approach to exploiting coarse-grained pipeline parallelism in C programs," in *International Symposium on Microarchitecture*, 2007.
[13] G. Ottoni, R. Rangan, A. Stoler, and D. I. August, " Automatic Thread Extraction with Decoupled Software Pipelining," in *International Symposium on Microarchitecture*, 2005.
[14] J. Cheng, J. Castrillon, W. Sheng, H. Sharwachter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, and H. Kunieda, "MAPS: An Integrated Framework for MPSoC Application Parallelization," in *Design Automation Conference*, 2008.
[15] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, et al., "A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS," in *IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, February 2010, pp. 108–109.