Adaptive Multi-Threading for Dynamic Workloads in Embedded Multiprocessors

Chenjie Yu  
University of Maryland, College Park, USA  
cjyu@umd.edu

Peter Petrov  
University of Maryland, College Park, USA  
ppetrov@ece.umd.edu

ABSTRACT

We present a framework for run-time parallelism adaptation of multithreaded applications in multi-core systems. Multi-core systems often execute diverse workloads of multithreaded programs with different system resource utilizations and varying levels of parallelism. As a result, the availability of system resources for individual components of the workloads changes at run-time in an unpredictable manner. Consequently, the level of statically determined parallelism by the system infrastructure, e.g., number of concurrent threads, could be suboptimal and lead to performance degradations. The proposed framework monitors the dynamically changing shared system resources, such as the available processor cores, and adapts the number of threads used by the applications throughout a parallel loop execution so as to match the parallelism level to the changed state of the system resources. The end result is the elimination of sizable overheads due to improper level of parallelism, and the resultant serialization of threads on a single core, that could easily occur in a dynamic system environment.

Categories and Subject Descriptors: C.3 [Computer Systems Organization]: Special-Purpose and Application-Based Systems


Keywords: Adaptive Multi-Threading, Multi-Core Systems

1. INTRODUCTION

Uniprocessor systems have encountered enormous design difficulties as processors reached GHz frequencies. Design complexity, circuit synchronization, speed gap between processors and memory, power consumption, and thermal issues have hindered the rate of further advancement. To continue the growth of computer system performance, the industry has turned to multi-core platforms [1]. Multi-core processors, which often consist of multiple but simpler cores running at lower frequencies, naturally address many of the above problems with promising and steady increases of theoretical peak performance [2].

While multi-core platforms offer new dimensions to exploit application parallelism, the two major challenges are that the software is required to expose a sufficient number of concurrent threads and, equally importantly, the underlying OS and multi-core hardware have to coordinate and manage their run-time execution. The potential of multi-core platforms cannot be fully realized without properly addressing both of these major problems. In this paper we focus on the latter problem, that is, even when a program is well parallelized, its performance can still be severely undermined due to frequent changes of system resources availability at run-time. Such resource fluctuations are often unpredictable and lead to partially serialized execution of parallel programs and overheads of over-parallelization, thus necessitating a dynamic method to address it.

Modern embedded applications employing multi-core platforms are usually comprised of multiple programs (possibly multi-threaded) which together with the OS share and compete for system resources, creating a dynamically changing execution environment [3]. Consequently, the availability of system resources for any individual program varies with time. Any one of the shared resources may become the bottleneck that implicitly limits parallel application performance and even lead to much greater overheads. This results in dynamically changing “optimal parallelism” for even well parallelized and well tuned programs that have not taken into account the dynamically changing system resources. In our experimental study we have evaluated the impact on performance of the traditional, non-adaptive multithreading, when the number of available cores does not match the fixed number of threads.

Single parallel tasks are normally coded to use a fixed number of threads. The level of parallelism is fixed or set at compile time or when the task starts execution. The number of threads to spawn is often determined by the amount of parallelism that can be exploited from the target application, and by the supported parallel processing capability of the underlying system platform, including multi-core hardware and operating system. For many multi-core parallel machines, the applications attempt to use all the available processor cores with each thread allocated to a core so as to gain the maximum speedup. While this mapping scheme simplifies task parallelization, the actual run-time behavior can be suboptimal due to the OS overhead and the interference from other tasks in the system. This is mostly due to multiple threads/tasks sharing resources on the same core, such as L1/L2 caches, TLBs, BTBs, etc. as well as resources at system level, such as memories and interconnect networks, which results in interference between applications and between threads of the same program.

Such a situation is illustrated in Figure 2, in which the cores allocated to task A, along with other associated resources, change due to the activity of other programs in the system. When the number of available cores, for instance, is reduced due to activities of other applications (or the OS), some threads will need to be allocated on the same cores, possibly with threads from other tasks. In this case,
even though the application has been coded for a high concurrency level, the actual execution sequence corresponds to a partially serialized execution, plus the cost of switching between threads and the overheads of parallelization itself.

The fundamental reasons for such problems are the resource contentions across applications, and their disparate resource utilization, which changes the actual parallelism that the system platform can truly offer. The actual optimal parallelism level is entirely application (workload) and platform dependent and is impossible to determine at compile/design time.

In this paper, we propose a framework that implements a run-time parallelism adaptation method in the presence of continuously changing availability of system resources. Among all the shared resources, we use the most important one, the available processor cores, to demonstrate the workings of this framework. The technique dynamically adjusts the target application to a proper parallelism level so that its efficiency approaches the optimal, corresponding to the specific system resource availability at any moment. The proposed technique is complementary and independent of the underlying OS scheduler. An on-line mechanism is employed, ensuring that the application adapts to the optimal number of threads, within a long-running parallel execution. Our experiments demonstrate both the inefficiencies of the traditional fixed parallelization approaches in the face of dynamically changing availability of processor cores as well as the improvements achieved by the proposed framework.

2. RELATED WORK

Several research projects have tried to address the problem of identifying the optimal level of parallelism for multithreaded programs on parallel platforms. [4] addresses the problem of finding the optimal number of parallel threads for the specific machine configuration at the time when the parallel program is to be executed. The optimal number of threads is determined at the very beginning through a run-time profile of the initial loop iterations. In [5], the authors study the potential of multithreaded platforms (Cray’s MTA and Sun’s Niagara) with a focus on data-intensive, irregular scientific applications, as well as other micro-benchmarks. Their study reveals that large amount of parallelism cannot be achieved with improper level of on-chip cache capacity and bandwidth resources. In [6], the Altix and the Cray Opteron Clusters are used as a platform for the NAS benchmarks. The results show how different machine configurations can have a dramatic impact on the performance. Severe memory contention easily occurs in massively parallel machines.

In [7], the authors propose a dynamic performance analysis approach (SelfAnalyzer) and a corresponding Performance-Driven Processor Allocation policy that distributes processors according to the performance measurements. [8] studies dynamic processor allocation policies for multiprogrammed shared memory multiprocessors. In [9], the authors propose an on-line adaptation mechanisms for NOC-based heterogeneous multi-processor systems that improves the system efficiency for hard to predict at design time execution scenarios. In [10], the authors utilize a compiler based adaptive parallelism method to adjust the number of processors that are active during execution. Power consumption is reduced by putting unused processors into low-power mode.

The framework proposed in this paper adapts and fine-tunes the parallelism-level at run-time in order to match the dynamically changing system resource availability. The dynamic change of resource utilization is monitored and the best suitable parallelism level is determined and enforced to the target program. In the following sections, we use the most critical source of contention, the available processor cores, to illustrate our idea. Other system resources could be integrated in the same framework as well.

3. RUN-TIME RESOURCE AVAILABILITY

In many parallel programming models, programs are parallelized so that each thread is assigned to a processor core. A thread is mapped to a dedicated core and its associated resources such as private caches, TLB, BTB, etc. The total number of threads is often equal or close to the number of processor cores, so as to exploit the maximum possible speedup on the specific parallel machine. While this is true in experimental simulations and cases where no system software or other programs need to be executed in parallel, such assumptions can be greatly undermined in reality where the program is executed on a machine with significantly different characteristics or when the system resources are shared by multiple programs.

The typical assumption of one thread per core in many parallelization methodologies frequently does not hold true when there are other programs executing in the same system. For example, for an eight core machine, a well parallelized FFT task can utilize all eight cores to speed up computation. However, with other user programs and OS executing simultaneously, the number of cores available to the FFT task can vary significantly at run-time. Thus, if initialized to run with eight threads, at certain times there will be multiple FFT threads and threads from other tasks sharing the same core, along with other resources; the OS or the thread library will need to switch contexts between them.

The performance degradation is a result of the combination of direct and indirect overheads. The direct overhead includes the cost...
of the OS performing context switching between kernel threads and/or thread library switching user level threads, as illustrated in Figure 2. Thread specific information needs to be saved and restored during this process. This depends heavily on the OS and thread library implementations. Nevertheless, such overhead cannot be ignored for applications with fine-grained and frequent synchronization operations. The intrinsic overhead of parallelization is rather application dependent. However, it typically includes more private data space with more threads, and more synchronization operations.

The indirect overhead comes as a result of interference in the locally shared resources between threads mapped to the same core. For example, L1 caches can be thrashed by the threads that work on different data regions. Whenever one thread is taken off and another is swapped in for execution, significant number of cache misses will occur due to data cache interference as they normally operate on largely disjoint data regions. Contrasted with the situation of a single thread per core exclusively using the local L1 cache, this can result in significant performance degradation, especially for systems with limited cache capacity and memory bandwidth. Other shared resources, such as TLB, BTB, etc., cause similar indirect impact. In our experiments we evaluate the extent of the D-cache interference and the overall performance overheads and demonstrate their significance.

4. DYNAMIC PARALLELISM ADAPTATION

The changing availability of system resources argues for an online adaptive method capable of monitoring the resource availability and identifying the optimal parallelism level with respect to the current system platform status. Figure 3 shows a segment of an application execution, during which the available number of CPUs changed from four to two, and then back to four. The left side shows a traditional parallel execution with a fixed number of threads that cannot be adjusted to such changes. The combination of fixed number of threads and changing number of cores results in the overhead to the total execution time as shown in the figure. On the right side, the application is able to detect and adapt to such situations and change the number of threads accordingly by utilizing the proposed technique. The application can only adapt its parallelism at certain time intervals, during which the system state is probed and a decision is made whether or not to change the parallelism level. Clearly, the granularity of adaptation determines how effectively the proposed method can track the dynamically changing environment.

It is important to note that the proposed parallelism adaptation is performed not only at the beginning of a parallel loop but also throughout the execution of such performance-critical application phases that support flexible parallelism level. The technique is independently applied to all the parallel loop/functions of the application, as in the more static adaptive parallelism methods [10]. Effectively the loop iteration space (the outermost loop) is partitioned into multiple pieces and the adaptation from one parallelism level to another is affected when transitioning between these pieces. We refer to each such partition consisting of identical functionality but executing on a different part of the original loop iteration space as an adaptation unit, while the points between them are referred to as adaptation points.

Several mechanisms are needed to implement the proposed framework. First, mechanisms are required to monitor and capture the resource availability. This information must be made readily available at the adaptation points in order to perform, if needed, a rapid adaptation to the parallelism level for the subsequent adaptation unit. Second, a simple and generic loop transformation is performed which based on desired granularity of adaptation and total number of loop iterations, partitions the loop iteration space into identical adaptation units. This transformation is somewhat similar in spirit to loop unrolling. Third, a code that determines and adapts the best parallelism level is needed. This decision making process will be executed at the adaptation points and it must satisfy a number of important requirements. The part of it that executes at runtime needs to be extremely fast. Adapting parallelism level must be achieved within tens of cycles, which is minimal compared to the actual adaption unit length, yet keeping the current level of parallelism (due to no changes in availability of resources) must cost no more than a few cycles. In this paper we focus on adapting to processor core availability to illustrate each of these fundamentals for the proposed technique. Nonetheless, this parallelism adaptation framework can be extended to other types of shared resources, such as interconnect bandwidth and cache memories.

4.1 Adaptation Units and Points

To be able to dynamically adjust application parallelism, the applications need to provide a number of cut-off “points” within a long parallel loop/function where the resource availability is checked and a decision is made as of what is the best level of current parallelism. The instantiation of such adaptation points can be achieved in a number of ways. Since the proposed technique is applied on the major application loops, these loops are the targets of a relatively simple transformation that allows the efficient definition of adaptation points. The loop is segmented into a fixed number of
smaller loop iterations each of which operates on a subset of the original iteration space, as illustrated in Figure 4.

The smaller loops into which the original loop is segmented represent the adaptation unit (AU). AUs are the actual forms of loop execution at different time segments. The AU is assumed to be parallelized using different levels of parallelism, i.e., several copies of the AU code exist each operating with a pre-determined number of threads. For instance, a two-threaded and a four-threaded versions of the AU may be present and chosen at different moments. The control algorithm decides which AU implementation to choose based on the resource utilization at the moment. The different AU versions can be obtained either through manual parallelization or by using a parallelizing compiler. This can be easily achieved with most compiler parallelized programs. In our experiment, we have used a number of Splash-2 benchmarks, which are written with parametrized number of threads and are very easy to adapt to different parallelism levels.

A special code is inserted prior to the AU - this code serves the role of the adaptation point. It rapidly checks system resource utilization and makes a decision based on the specific application and machine configuration as of whether or not a change of parallelism is required. This decision making involves various trade-offs and needs to be performed in an extremely efficient manner. Moreover, several implementation approaches can be followed to implement the adaptation functions with varying involvement of hardware and software. One such implementation, which is mostly software-based is described in the next subsection.

The structure of the adaptation logic is illustrated in Figure 5. When the logic determines that there is a need to change parallelism level, it selects from the pool of parallel AU implementations that best fits the current scenario and resumes execution with that implementation until the next adaption point. The cost of this is identical to the one of passing a function pointer. The decision making process could cost more cycles, based on the control algorithm used to track optimal parallelism. In our experiment, we have used simple look-up tables to store off-line knowledge of matching information so that the run-time cost is minimum.

More complex dynamic algorithms can still be beneficial, however, as long as the benefit from adaption outweighs the overheads. This can be combined with the trade-off of the AU segments length. Longer AU segments allow for fewer adaption points. In our experiment, we have used the adaption logic cost to be within tens of cycles. This cost is materialized only when there is a need to change parallelism. By doing this, the overhead from the adaption logic itself is hidden, which allows for superior performance improvements.

The granularity of adaptation, i.e., the frequency of adaption points, is another issue involving trade-offs. Clearly, if the system adapts slowly, the application may be executing with deteriorated performance for long periods of time with less parallelism when more cores are actually available or vice versa. This depends on the specific workloads and the specific platform. In this paper, we are targeting long parallel execution loops. Our adaption logic implementation captures major resource changes as it is executed at several thousands of cycles granularity level.

4.2 Online Adaptation Algorithms

Initially, each core is assigned one thread (such a mapping is often optimal as is seen from our experiments). However, there are cases when such mapping may not be optimal depending on different application requirement. When the threads execute long-latency operations, such as I/O operations, frequently, it may be more efficient to map more then one thread per core so as to hide the long I/O latencies with useful computations. Furthermore, some applications can only utilize an even (or power of two) number of processors, such as most of the Splash-2 programs. When an odd number of cores are made available, the adaption logic must rapidly select the optimal number of threads by having some of the parallel threads share the cores. Such knowledge is very application and system platform specific and can be obtained through profiling. In any case, this selection of the optimal number of threads can be very efficiently implemented through a small application-specific table indexed by the adaptation logic with the number of available cores.

The cost of such adaption also involves a direct part and indirect part. The direct part is very limited, which only involves a look-up into a predefined table and switch to another predefined parallelized implementation. The indirect part, which includes cache misses due to change of parallel implementation, is difficult to analysis. However, in our experiments, the combined overhead is contained within less than 0.1% of execution time.

In general, a resource tracking framework needs to check a number of potential system bottlenecks at the same time. Due to interference and over-utilization each one of them could prevent the applications from further scaling up its parallelism level. In such a case, a more complex control algorithm will be needed that possibly consults several tables. In this paper, however, we have focused our attention on the problem of adapting to the core availability.

5. EXPERIMENTAL RESULTS

We have conducted extensive experiments to evaluate the proposed adaptive multithreading framework. We have used the cycle accurate full-system simulator M5 [11]. The simulator has been extended to support multithreaded execution in system call emulation mode, with context switching cost reduced to minimum. The machine configuration consists of up to 32 cores; each core has a 16K L1 cache and all of them sharing an 8M L2 cache. The simulated platform is rather conservative in terms of system overhead, as it only models the thread library overhead. Real system executions employing embedded or other operating systems would involve significantly more direct overhead related to context switch.

We have used Splash-2 benchmark [12] kernels and applications (RADIUS, LU, OCEAN), as well as matrix operation kernels, MMUL-matrix multiplication and SOR regression, and some image processing kernels: BLUR and EDGE from the GIMP open source image processing software, which are parallelized with parametrized number of threads.

We start our experimental study by analyzing the overhead of the traditional non-adaptive multithreading in dynamic environments where core availability changes. For this, we have performed sim-
Figure 6: Performance overhead for a 16-threaded implementation on a 8-, 4-, and 2-cores platform

Figure 7: Performance overhead for a 32-threaded implementation on a 16-, 8-, 4-, and 2-cores platform

Figure 8: D-Cache miss increase (%) of 16-threaded implementation on a 8-, 4-, and 2-cores platform

Figure 9: D-Cache miss increase (%) of 32-threaded implementation on a 16-, 8-, 4-, and 2-cores platform

Simulations on our benchmarks with fixed number of threads (16 and 32), while the number of available cores is set from 16 down to 2 covering all power of two cases. Figure 6 shows the performance of 16-threaded execution of all benchmarks on 8, 4, and 2 available processors. The performance is measured with the number of CPU cycles (maximum across all the participating processors). The bars represent the percentage increase in cycles when compared to the one-thread per core mapping for the same problem size. The left bar represents the percentage increase in cycles for 16 threads with 8 available cores - the comparison is against 8 threads on 8 cores; the middle bar is for 16 threads with 4 available cores compared to 4 threads on 4 cores; the right bar represents 16 threads running on 2 available cores, compared to 2 threads on 2 cores. All the evaluations and comparisons are performed for a fixed problem size. Figure 7 reports an identical study but for a 32-threaded execution on 16, 8, 4, and 2 available cores. Similarly, the bars represent the percentage increase in cycles compared to the one thread per core mapping. Note that our setup includes only threads belonging to the same program competing with each other. Such threads often share portions of the global data and instruction cache, which reduces the interference between them. The most notable overhead comes from data cache thrashing. When they compete with threads from other programs, the incurred overhead will be more significant.

A clear trend can be seen that performance overhead increases with the number of threads mapped to a single core. The performance impact as compared to the one thread per core mapping differs largely across the benchmarks, which is due to the differing overheads in terms of synchronization and interference in local caches.

To further understand these overheads and more specifically the overhead in terms of contention in the local D-caches, we show the impact (increases) on L1 D-cache misses for the same configurations as described above. Figures 8 and 9 report the increase in misses (in percentages) for the 16-threaded and the 32-threaded cases. Similarly, the comparison is against a one thread per core mapping. It can be seen that misses increase when more threads are mapped per core, which directly translates into a performance overhead that can be correlated with the negative performance impact presented above.

The execution in dynamically changing environments can be much more complex, as can be seen from the large variance in performance overhead when the number of available cores changes. Some benchmarks, like MMUL and LU do not suffer from sizable local D-cache interference as the working sets of threads sharing a core can coexist in the cache without much contention. Other factors, such as the overhead of parallelization itself versus a more serialized version, including synchronization, sometimes play a more dominant role. In this case, scaling to smaller number of threads improves the overall system performance. Nonetheless, the underlying premise of the proposed framework that improper level of parallelism causes performance degradations firmly holds.

To estimate the performance benefits of the proposed technique, we have defined two execution scenarios modeling a change in the number of available cores. For the baseline case we assume
that a fixed number of threads are used throughout the benchmark execution. The thread library manages the allocation of threads to available cores. We contrast this with the proposed technique where the change in core availability is tracked and the number of threads adapted.

Figure 10 shows the core availability for the two environments that we have evaluated. The y-axis represents the number of available cores, while the x-axis represents the elapsed execution time in percentages (execution completion occurs at 100%). The scenarios start and end with 16 and 32 cores, respectively. The baseline execution for the two scenarios consist of a fixed 16 and 32 threaded version of each benchmark, respectively.

Figures 11 and 12 report the achieved performance improvements. The bars represent the reduced execution cycles (in percentage) when compared to the baseline. The benefits vary across the benchmarks, as well as across the specific scenarios they are executing in. As expected, the benchmarks that are more sensitive to mismatched parallelism overhead benefit more from the proposed technique. Performance improvements of up to 36% can be observed for both scenarios.

6. CONCLUSIONS

In this paper, we have addressed the problem of execution efficiency of multithreaded programs in multi-core platforms. We show that system resource availability, such as processor cores, can significantly impact the performance efficiency of multithreaded programs. We have proposed an online parallelism adaptation framework capable of adjusting the level of parallelism to dynamically changing availability of processor cores. Our experiments demonstrate significantly improved execution efficiency as a result of tracking and adapting to dynamically changing resource availability.

7. REFERENCES