

A Practical Approach for Reconciling High and Predictable Performance in Non-Regular Parallel Programs

Olivier Certner, Zheng Li, Pierre Palatin, Olivier Temam
Alchemy Project
INRIA Saclay
France

Frederic Arzel, Nathalie Drach
Laboratoire d'informatique de Paris 6
Pierre et Marie Curie University (Paris 6)
France

E-mail: {olivier.certner, zheng.x.li, olivier.temam}@inria.fr

Abstract

Increasingly complex consumer electronics applications call for embedded processors with higher performance. Multi-cores are capable of delivering the required performance. However, many of these embedded applications must meet some form of soft real-time constraints, and program behavior on multi-cores is even harder to predict than on single-cores. In this article, we highlight the greater performance variability of irregular applications (non-regular control flow and/or data structures) across data sets when parallelized and run on a multi-core. We then show that a proper parallelization approach coupled with a lightweight run-time system can drastically reduce this performance variability without sacrificing their performance. This approach requires no complex program or architecture analysis or modeling. Moreover, we show that parallel program performance becomes stable enough that it is possible to reasonably and accurately predict it by sampling a few training runs.

1. Introduction

Until recently, the necessity to predict the execution time of an application was essentially a feature of real-time systems. These systems, in turn, were associated with simple controllers or elementary processors running similarly simple applications. Due to both the increasing performance of embedded systems, and the consumers' taste for rich and complex applications, many programs are now more complex, must achieve high performance and their performance must remain reasonably predictable. They range from soft real-time mobile applications such as GPS navigation software, to consumer electronics or desktop applications such as video compression/decompression, games, rendering software, or even scientific applications such as real-time finite-element modeling for engine control [4].

Two factors determine the predictability of a program execution time: the program workload (depending on the algorithm and the input data set) which is architecture-independent, and the program behavior on the architecture. While the impact of the first factor is not trivial, one can correlate in many cases a program execution time on a single core to some restricted set of data characteristics or program parameters. However, the execution time variability on multi-cores is significantly higher than on single-cores due to thread partitioning, balancing issues and contention. As a result, the second factor is becoming both important and tedious to ad-

dress. Therefore, just as applications become more complex, and the performance of an increasing span of applications must become predictable, the widespread use of multi-cores will make this task even harder. The purpose of this article is to focus on this second factor, and to show that, through a proper parallelization approach, it is possible to get parallel programs with fairly predictable throughput, with neither compromise on speedup nor ease of programming.

In real-time systems, performance prediction is today essentially based on a detailed knowledge of the underlying architecture, and the program behavior on this architecture. The approaches can be split into (sometimes combined) three categories. (1) Trying to predict the detailed performance of programs on architectures, such as Absint [12] for simple pipeline architectures and for superscalar architectures [8]. Or (2) changing the program so that its behavior is better understood and predictable, such as StreamIt [5] for stream processing applications. Or (3) changing the architecture so that its behavior is more predictable [10], such as disabling the cache or replacing it with scratchpads, using software instead of hardware cache coherence, VLIW instead of out-of-order execution, etc.

In summary, most approaches rely either on analyzing in details program behavior on architectures, or on simplifying programs or architectures. The former is increasingly difficult as architectures become more complex, and the advent of multi-cores will render this approach even harder. The latter can have the effect of reducing program performance, or the transformations are domain-specific and cannot benefit most applications.

In this article, we propose a fourth approach for achieving both easier to predict and high performance on multi-cores/multi-threaded architectures. For complex applications with irregular behavior, one of the main sources of performance variability on multi-cores is simply that not all cores will be used at any time. Such poor load balancing can induce both irregular and poor performance. Our approach is then based on a simple principle: using run-time conditional task division, we can design parallel programs, even (especially) programs with irregular behavior, so that they try to leave no processor core unused at any time. That cores are used most of the time makes it easier to deduce performance based on a sample of the execution, for computationally intensive pieces of code.

This approach requires no knowledge of the architecture, and thus no detailed analysis. It is not tied to any particular way of handling concurrent accesses, thus allowing programmers to use it in combination with, for example, mechanisms that avoid deadlocks, such as the recently popular transac-

tional memory approach [6]. It can also adapt to varying architectures and scales well with the number of cores. Finally, it can be applied efficiently and independently to any computationally intensive part of a program.

The approach is based on the CAPSULE programming environment [9], which blends well with the popular component-based programming paradigm. Its principles are to ease parallel programming and to maximize cores usage, which are briefly reminded in section 3.1 along with an example of use. Like Cilk [2], or to a lesser extent Charm++ [7], CAPSULE implements parallelization through the intuitive operation of splitting/dividing an encapsulated task in two. Unlike the aforementioned environments, this division is *conditional* upon available hardware resources, in order to match task granularity to available resources and to considerably reduce the overhead associated with very small tasks. The runtime systems *probes* the hardware to decide whether it actually carries through the division. We improve upon the original CAPSULE approach, and we show that:

1. While a hardware support was required for implementing fast probing [9], we can now achieve the same speed using a software-only speculative approach. Fast probing is critical to rapid adaptation to hardware availability, and therefore, to performance stability. The probing software implementation readily opens up the approach to a large range of existing processors.
2. Using this run-time (dynamic) parallelization approach, the execution time of irregular programs is not only lower but also much more stable than statically parallelized programs (sections 2, 4 and 5). As a result, these implementations lend well to performance prediction.
3. Performance prediction can be achieved through iterative execution time sampling: the program is executed on one or several data sets, and the resulting execution time is used to predict program throughput (sections 3.3, 4 and 5).

One can observe that the behavior of division-parallelized programs is even more complex (run-time dependent) than the behavior of statically parallelized programs. However, we show that their performance is more stable and thus more predictable thanks to dynamic adaptation. This observation means that accepting to relax control and prediction of the detailed program behavior can paradoxically result into better prediction accuracy without loss of performance.

2. Motivating example

Consider the example of the Quicksort sorting algorithm, where an array is recursively split into two sub-arrays according to a pivot element. An intuitive but naive approach for a 2-way parallelization of Quicksort is to perform the recursive sorting on the first two sub-arrays concurrently. By repeatedly doing so for subsequent sub-arrays, on an N cores machine, there are enough sub-arrays for each core after $\log_2(N)$ pivot steps, and all hardware resources are used at that point.

However, this parallelization is static, and the parallel program performance will be quite data set dependent because two sub-arrays can have very different sizes. As a result, the workload of each core can vary considerably from one input array to another. Some cores will finish their work sooner than others, and will be left idle. Figure 1 shows an example of workloads for a random array of 2000 elements on a

4 cores machine. The initial array is represented as the node on the left. A node's children represent the sub-arrays to be sorted independently on different cores after one pivot step. The number on a node indicates the sub-array size.

Figure 3 (4 cores - static) shows the performance of static parallelization for 1000 random arrays of 1M elements. One can observe that the variability of the execution time on multi-cores is much higher than that observed on single-core machines. In short, parallelization increases execution time variability, or conversely decreases program execution time predictability.

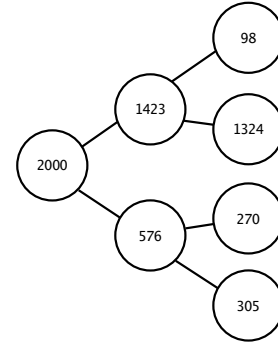


Figure 1. Static parallelization of Quicksort on a 4 cores computer.

Now, let us assume the program is parallelized the same way, i.e., two sub-arrays are treated concurrently, except that this parallelization is done at every pivot step, regardless of the number of cores. The cons are that the task granularity will become exceedingly small after all possible pivot steps, with leaf sub-arrays of one element each, voiding the benefits of parallelization. The pros are that the potential number of tasks, and thus the degree of parallelism, is very large.

In order to get the best of both worlds, this systematic parallelization is in fact performed *conditionally* upon available cores in the CAPSULE programming model. At every pivot step, if a core is available, the Quicksort program will parallelize the treatment of the two resulting sub-arrays. Otherwise, it will sort them sequentially. As a result, cores will almost always be used, but the task granularity does not risk becoming too small, except towards the end of the execution.

The pattern of division depends both upon the data set and the cores occupation (possibly even by other processes). Figure 2 shows the execution of Quicksort parallelized that way for the same array as in Figure 1. Each graph node indicates that a parallelization (division) occurred, i.e., one sub-array has been assigned to a core for execution. One can note that more divisions occur on the path with the highest workload (1423) because it takes advantage of the cores freed early by the small-workload path (305). The irregularity of the graph reflects the difficulty of predicting when each core will become available. Figure 3 (4 cores - dynamic) shows the performance of the dynamic parallelization for the same data sets as for the static parallelization.

Now, the key observation is not so much that the performance of this CAPSULE-parallelized version is better than the statically parallelized version on average, but that, over 1000 arrays, the performance of the CAPSULE-parallelized version is far more stable than the performance of the statically parallelized version. Let us pick 10 random arrays, compute the average execution time for both versions, and

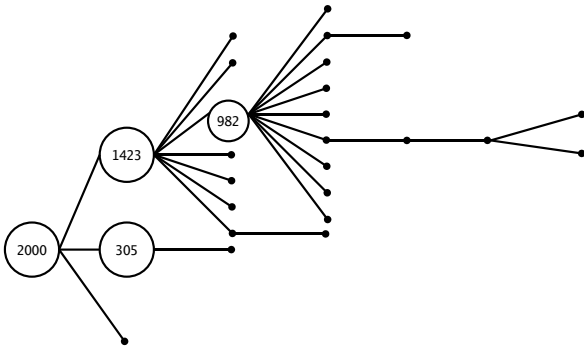


Figure 2. Dynamic parallelization of Quicksort on a 4 cores computer.

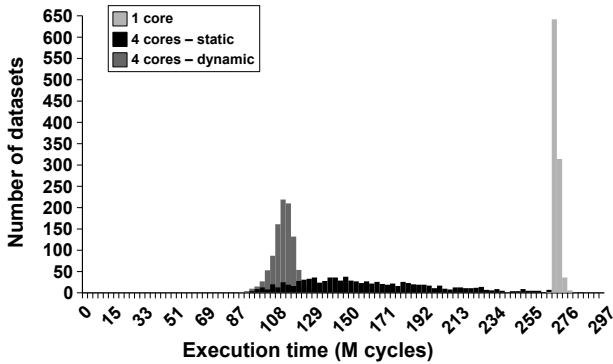


Figure 3. Performance of static vs. dynamic parallelization of Quicksort.

then compare it to the average execution time over 1000 arrays. The error for the statically parallelized version is 6.42%, while the error for the CAPSULE version is 1.21%. In fact, a sample of 3 arrays only is sufficient to achieve an error of 2% or less with the CAPSULE version versus 100 arrays for the statically parallelized version, as shown in Figure 4.

This example suggests that it is possible to precisely predict the behavior of irregular program parts using a combination of conditional parallelization and execution time sampling.

3. Program Parallelization Through Division for High and Predictable Performance

In this section, we first recapitulate the principles of the CAPSULE parallel environment. Then, we introduce the new software-only probing mechanism which opens up conditional division parallelization to a broader range of processor architectures. Finally, we propose a methodology to predict performance which relies on this environment.

3.1. Parallelization Through Division

A key principle and advantage of the CAPSULE environment is that a programmer does not need to have any information on the underlying architecture to be able to write efficient parallel code. He is simply provided with a very abstract form of architecture, composed of an unlimited number of cores,

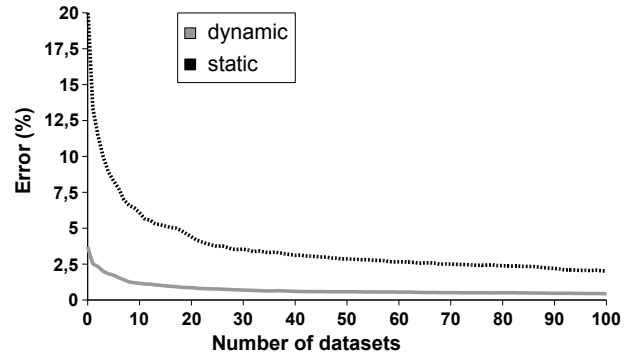


Figure 4. Predictability error (static vs. dynamic).

in which he only needs to specify where a task can be split in two, and how the remaining work is to be distributed among the two tasks after division. By doing so, he essentially indicates to the environment *potential* parallelization opportunities.

Concretely, the programmer can issue *probe* requests, by calling the `capsys_probe` primitive. He can do so even within innermost loops without worrying about overheads (see section 3.2). The environment replies to these requests by indicating if another core is available to execute the new task. In case of a positive reply, the division is actually carried through by the environment after the `capsys_divide` primitive has been called. In the other case, the programmer schedules the task to be executed sequentially. Figure 5 shows a sample of code from the Quicksort example. This code is an excerpt of the `qs_base` function that performs the sort when both sub-arrays are non-trivial. `qs_capsule_wrapper` is a wrapper function of `qs_base` used for argument conversion purposes.

```
context_t * ctxt = NULL;
// Probe hardware for available core
ctxt = capsys_probe (qs_capsule_wrapper);
if (ctxt)
    // Perform division and quicksort on the
    // right sub-array
    capsys_divide (ctxt, (void *) alloc_qs (
        right + 1, end_idx, array));
else
    // Division denied, carry on sequentially
    qs_base (right + 1, end_idx, array);

// Quicksort on the left sub-array
qs_base (begin_idx, right - 1, array);
```

Figure 5. Quicksort example code.

Porting a sequential program to the CAPSULE environment requires a similar effort as porting it to popular threading environments. Porting an already parallel program, in order to use the CAPSULE API instead of a threading API, is straightforward. Exploiting the dynamic parallelization provided by CAPSULE, however, requires some knowledge of the program's inner workings and some changes in the most computationally intensive parts.

3.2. Low-overhead Probe and Divide

Given that the programmer will issue very frequent probe requests in order to indicate all possible parallelization opportunities in his application, such requests must have a very low cost for this approach to be efficient. The first version of CAPSULE relied on a hardware support to achieve 1-cycle probes on an SMT [9]. The new software-only version of CAPSULE reaches the same average performance, i.e., 1-cycle probes, on an Intel Core 2 Duo processor, which allows probe inclusion within innermost loops¹.

Low-overhead software probe is achieved using a speculative technique bearing similarity to the test-and-test-and-set protocol [11]. A counter indicates the number of currently available cores. When the program calls the `capsys_probe` primitive, the run-time system checks this counter. Instead of going into the full process of atomically reading and eventually decrementing the counter to account for the reservation of a new core, the CAPSULE run-time system first uses it as a hint. It reads it without synchronization, and refuses the division request if the counter value is 0. For other values, the run-time systems follows the usual process.

When the division is to be refused, this technique avoids the overhead of synchronization most of the time. As a result, speculative probing requires 1 cycle on average, while accurate lock-based probing requires 121 cycles. When a probe succeeds, the overhead still exists, and can be accounted as part of the actual division process. Overall, this overhead remains low because there are much fewer divisions than probes.

Using the counter as a hint is done by a simple test and branch instruction at the assembly level. Since this operation does not generate any data dependency and since division is refused most of the time, the branch prediction mechanism hides the cost of the test, which is why software-based probes are almost as efficient as hardware-based probes.

3.3. An Iterative Approach for Predicting Parallel Execution Performance

The more stable throughput of programs parallelized using the dynamic division mechanism of CAPSULE can be leveraged for predicting program performance by sampling executions. For that purpose, we use principles similar to iterative compilation ones [3], which relies on multiple executions of the same program to fine-tune optimizations. In our case, the multiple training executions are used to estimate the average throughput and the throughput relative dispersion of a program across data sets, and then to forecast future execution throughputs.

Each training run, numbered i , uses one distinct input data set, and produces a throughput measure t_i . After n runs, the estimated throughput average $\widehat{\mu}_n$ is:

$$\widehat{\mu}_n = \frac{\sum_{i=1}^n t_i}{n}$$

The estimated relative dispersion sequence is, at this point, defined as:

$$\widehat{disp}_n = \frac{\sum_{i=1}^n |t_i - \widehat{\mu}_n|}{(n-1)\widehat{\mu}_n}$$

¹On older Intel and AMD processors, we have seen this cost vary from 1 to a maximum of 10 cycles, on average.

The training phase lasts until the estimated average and dispersion both converge to the actual average and dispersion values. Taking the example of the estimated average, it is useful to introduce the sequence of relative differences of consecutive estimated averages, i.e., at step i , the value:

$$\frac{|\widehat{\mu}_{i+1} - \widehat{\mu}_i|}{\widehat{\mu}_i}$$

since this sequence converges towards 0.

The operator has to stop the process when the estimators have reached the desired precision. Many statistical tests exist that can affirm with a chosen probability that these estimators are accurate enough to explain the experimental results, such as the well-known Student's test. For our experiments, we have chosen a much simpler empirical approach: we ended the training phase when 5 elements in a row were below a 1% threshold for both of the relative difference sequences (estimated averages and estimated relative dispersions).

Throughput vs. Execution Time. For programs where single-core execution time variability depends on known data set characteristics, e.g., the array size for Quicksort or the matrix dimension for Matrix-Vector Multiply, it is possible to predict parallel execution time, not just throughput, assuming all training data sets have the same characteristics.

When single-core execution time is a complex function of data sets, the prediction can still be achieved if a statistical model of execution has been built from experiments. In other cases, the prediction is restricted to program throughput, i.e., the average program performance per unit time² cumulated over all the cores.

4. Experimental Framework

Unlike for single-cores, there is no widely accepted parallel benchmarks suite for multi-cores, except for the Splash benchmarks. However, the latter are scientific computing benchmarks, which often have a regular behavior and which are not always within the scope of real-time applications. A recent survey on parallel computing and benchmarking [1] suggests the "dwarves" approach where, instead of large applications, the different characteristics of a parallel machine should be exercised with a mix of targeted benchmarks. To a large extent, we have been following that approach by progressively building a mix of kernels and larger applications, all parallelized using CAPSULE. This suite includes kernels and more complex applications, with both known regular or irregular behavior. The benchmark list is indicated below, together with a short description of the data sets used.

MxV The standard Matrix-Vector Multiply kernel. Its performance is obviously influenced by matrix size only. 1000 matrices were randomly generated.

SpMxV A Sparse Matrix-Vector Multiply kernel, where matrices are expressed in the Harwell-Boeing format. An increasing amount of real-time applications rely on physics modeling (games, engine control, etc), which can rely upon sparse matrices. 300 matrices were collected from MatrixMarket, a web resource for test matrix collections. 200 matrices were randomly generated using a random sparse matrix generator called Matgen.

²Commonly expressed in MIPS.

Dijkstra The graph shortest path algorithm used in network routing or for navigation purposes. 200 graphs were randomly generated with 1000 nodes and 2000 edges.

QuickSort The efficient and common sorting algorithm, tested with 1000 arrays of 1M elements.

Watershed An image segmentation program used for identification of image areas. The input images sizes ranged from 512×512 to 1536×1024 pixels.

GZIP The popular file compression utility. The storage on embedded devices being scarce, compression is commonly used for example on smartphones and PDAs. 500 files of different content types (audio, video, image, office document, telecom, etc) were used.

X264 A parallel implementation of an H264/MPEG4 encoder using slicing. Inputs were 134 clips of 100 frames with 640×360 pixels size.

The first 4 benchmarks were written from scratch with the CAPSULE API. The number of lines of code, for both sequential and CAPSULE versions, is several hundreds, depending on the benchmark. The watershed code was provided by a company doing some image processing, and was substantially rewritten to exercise the dynamic parallelization of CAPSULE ($\approx 50\%$ more code lines). The Dijkstra algorithm was parallelized as described in [9].

The GZIP and X264 benchmarks, by contrast, were converted to CAPSULE by a simple substitution of the POSIX threads calls by corresponding CAPSULE ones. When these original benchmarks are run, they initially create a number of threads corresponding to the number of available physical cores, and dispatch fixed workloads to them. Introducing dynamic division would have required to change some of the main algorithms used, which would have led to a major and costly rewrite of these applications.

All experiments were conducted on a bi dual-core machine equipped with 2 AMD Opteron dies and 4 gigabytes of RAM, half of them being attached to one of the two die in a cache-coherent NUMA architecture. The software-only CAPSULE has also been successfully ported to a 4-core ARM11 MPCore without operating system. Because the latter ARM platform allowed fewer test applications, we chose to perform experiments on the AMD platform.

5. Performance and Predictability Evaluation

Performance and Adaptation of Division-Based Parallelization. Figure 6 illustrates the speedups and scalability (see `dynamic`) achieved by CAPSULE. Even irregular programs, which are typically difficult to parallelize, such as QuickSort or Watershed, can take full advantage of an increasing number of cores.

The QuickSort and Dijkstra benchmarks, with their 2.43 and 1.58 speedups for 4 cores, scale the worst of all. For the former, this is because some divisions occur with too a small input array. Results can be greatly improved if no division is asked for sub-arrays under a given threshold (new speedup of 2.97 with a 100 elements threshold). For the latter, the increased contention compared to the static version is responsible for the sub-optimal result. Nonetheless, this is partly counterbalanced by the variability results for both versions below, when it comes to forecasting an execution time. For SpMxV, perfect speedup is not achieved essentially because

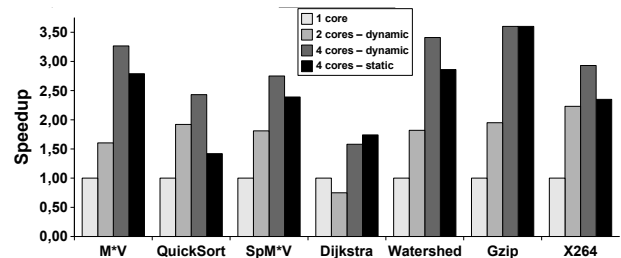


Figure 6. Speedup and scalability of CAPSULE parallelization.

of memory contention, but the result of the dynamic version is still significantly better than the static version's one.

Performance variability of parallel programs. Figure 7 illustrates the performance variability of static versus dynamic parallelization. The variability is measured through the relative dispersion (defined in Section 3.3) across all data sets.

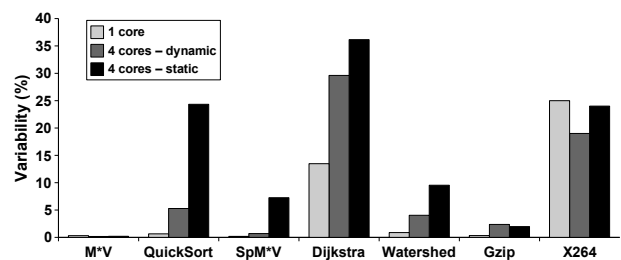


Figure 7. Variabilities of static vs. dynamic parallelizations.

The MxV and GZIP benchmarks exhibit a fairly stable throughput, whereas Dijkstra, QuickSort, SpMxV, Watershed and X264 have an irregular behavior across data sets. One can observe that, for the latter ones, the variability of execution time on multi-cores is often much higher than on single-cores. Our approach reduces it significantly and consistently, although it remains greater than the latter. Together with the increase in performance brought by CAPSULE, this feature opens up the possibility of trading off performance variability versus execution time in a more efficient way.

The reason why GZIP and X264 behave similarly for both parallel versions is due to our implementations, which don't allow them to really benefit from the dynamic division scheme (see section 4). The X264 CAPSULE version yet performs better than the static version because the latter destroys and recreates threads each time it processes a new frame, while the former, through the CAPSULE run-time system, reuses threads from a pool. Only GZIP performs slightly better in its static version. The difference accounts for the hardly higher overhead of CAPSULE compared to POSIX threads.

Predicting program performance. We now illustrate that the greater stability of dynamic division enables a rather accurate estimate of program performance by sampling a few data sets and using the iterative technique proposed in Section 3.3.

Figure 8 gives the number of data sets for which the mean execution time estimation is likely to differ by less than 5% from the real mean execution time, and shows that, for the irregular benchmarks, dynamic division can reach a given mean estimation error with fewer runs. Figure 9 illustrates the mean

estimation error when the prediction is based on a fixed number of data sets. It again shows that the lower dispersion of dynamic division results in a more accurate prediction using the same number of runs.

Dynamic versions bring a huge improvement for benchmarks that exhibit a high variability. The numbers for X264 and GZIP are almost identical for both versions, since the conditional division mechanism is not used. Results for Dijkstra are less spectacular because of contention, but still significant.

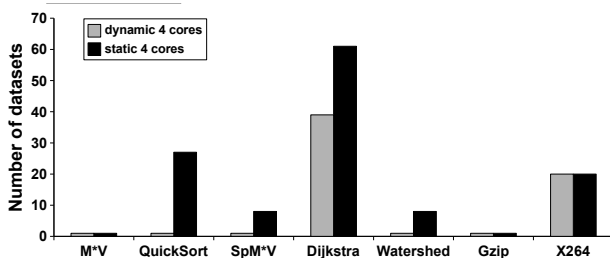


Figure 8. Number of data sets to reach a mean estimation error of 5%.

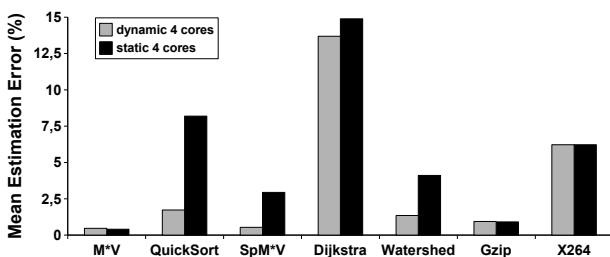


Figure 9. Mean estimation error after 10 runs.

6. Conclusion And Future Work

In this article, we have outlined the greater performance variability of irregular programs parallelized in a classic, i.e., static, manner. Unlike the original approach which relied on hardware support, we have proposed a software-only but similarly efficient dynamic parallelization technique, compatible with many architectures, and which brings several benefits. It improves performance and scalability by conditionally dividing running tasks to take advantage of available cores at any time during the execution. We have shown that, thanks to this property, the program performance becomes more stable across data sets. Moreover, we have proposed an iterative technique to leverage this property in order to predict parallel program performance. Programmers are thus better able to trade off execution variability for increased performance.

Trying to propose an efficient scheme to avoid too small divisions, while retaining platform independence, is an interesting challenge that would lead to even increased performance and less variability. Whether the scheme we already proposed in [9] is still applicable to the software version remains to be tested.

Acknowledgments

This work was sponsored by the French Research National Agency (ANR) and by the SARC European project. Olivier Certner is also funded by ST Microelectronics.

References

- [1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.
- [2] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, 1995.
- [3] J. Cavazos, C. Dubach, F. Agakov, E. Bonilla, M. O'Boyle, G. Fursin, and O. Temam. Automatic performance model construction for the fast software exploration of new hardware designs. In *International Conference on Compilers, Architecture, And Synthesis For Embedded Systems (CASES 2006)*, pages 24–34, Seoul, Korea, October 2006. ACM Press.
- [4] C. Dufour, J. Belanger, S. Abourida, and V. Lapointe. Fpga-based real-time simulation of finite-element analysis permanent magnet synchronous machine drives. In *Proceedings 38th Annual IEEE Power Electronics Specialists Conference, PESC'07*, June 2007.
- [5] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 291–303, New York, NY, USA, 2002. ACM Press.
- [6] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, page 102. IEEE Computer Society, June 2004.
- [7] L. V. Kale and S. Krishnan. CHARM++ : A Portable Concurrent Object-Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 91–108. ACM Press, Sept. 1993.
- [8] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *RTSS '99: Proceedings of the 20th IEEE Real-Time Systems Symposium*, page 12, Washington, DC, USA, 1999. IEEE Computer Society.
- [9] P. Palatin, Y. Lhuillier, and O. Temam. Capsule : Hardware-assisted parallel execution of component-based programs. In *The 39th Annual IEEE/ACM International Symposium on Microarchitecture, 2006*, Orlando, Florida, december 2006.
- [10] I. Puaut and C. Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 1484–1489, New York, NY, USA, 2007. ACM Press.
- [11] L. Rudolph and Z. Segall. Dynamic decentralized cache schemes for mimd parallel processors. In *ISCA '84: Proceedings of the 11th annual international symposium on Computer architecture*, pages 340–347, New York, NY, USA, 1984. ACM.
- [12] J. Schneider and C. Ferdinand. Pipeline behavior prediction for superscalar processors by abstract interpretation. In *LCTES '99: Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems*, pages 35–44, New York, NY, USA, 1999. ACM Press.