Futures and the Lazy Task Creation for .NET

Matko Botinčan
University of Zagreb
Email: mabotinc@math.hr

Davor Runje
University of Zagreb
Email: davor.runje@fsb.hr

Albert Vučinović
University of Zagreb
Email: ilijavanil@gmail.com

***Abstract*— This paper presents architecture overview of a .NET runtime for scheduling and executing futures on modern multiprocessor machines that is based on the lazy task creation technique. The runtime is responsible for efficient load balancing of fine-grained tasks on platform's processors and allows any function accessible from .NET to be called as a future.**

I. Introduction

The potential concurrency of many programs is inherently finer-grained than the concurrency of the platform they are executing on, i.e. the number of processors in the platform is typically much smaller than the number of potentially concurrent tasks in the program. Here (as well as elsewhere in the further text) a processor denotes a logical processor in the platform, e.g. it may refer to a core in a multi-core processor, as well as to a single core processor in a multiprocessor platform. A future is a simple abstraction mechanism that allows a programmer to expose the potential concurrency of such programs. The key challenge in achieving a scalable performance of programs annotated with futures is efficient partitioning and scheduling of the exposed potentially concurrent tasks regardless of characteristics of the underlying platform.

This paper presents architecture overview of the futures runtime for .NET that is based on a technique called *lazy task creation*. The runtime is responsible for efficient load balancing of fine-grained tasks on platform's processors. It is designed in such way that it allows any function accessible from .NET to be called as a future. We also introduce the notion of guards. The described constituents give rise to a simple extension of C# that incorporates programming language constructs for easy and elegant programming with futures.

A. Related Work

The term *future* was coined in 1977 in a Baker and Hewitt's paper [14], where they use it in discussing garbage collection issues in eager evaluation of subexpressions in a Lisp-like applicative programming language. The future as a programming language construct first appears in Halstead's work on MultiLisp [15], a version of the Lisp dialect Scheme which uses future as a principal construct for expressing parallel tasks. Later on, futures have been employed for integrating asynchronous remote procedure call mechanism in the context of distributed computing [6], [10], [17]. While [17]'s concern is extension of Argus by use of a dedicated communication mechanism called call-streams, [6] uses a more liberal message-passing environment for a general object-oriented language. [10] gives the first implementation of futures within the C++ programming language.

The first non-naïve approach addressing the problem of scheduling and executing futures is *load-based inlining* [16]. The basic idea is to create a new task computing a future only if the system load permits it, i.e. if there exists an idle processor. Otherwise, the future becomes inlined. In this settings, however, inlining of futures cannot be revoked, thus some processors may be assigned large tasks, while the others stay idle. Also, it is wiser not to create a separate task for every future, but to implement futures as "passive" objects containing enough information for carrying out their computation, which are then picked up and executed by a number of workers. However, an extra care must be taken in this case in order to avoid possible deadlocks.

Mohr et al. [18] address these issues by an elaborate technique called *lazy task creation*. In this approach, each future is evaluated in the current task, and the continuation of the calling function is moved to a separate task if some processor becomes idle, which in turn makes inlining of futures revokable. This way the program is executed sequentially until a parallel execution is possible — a principle reused by many systems targeting fine-grained concurrency (e.g. [5], [11], [12], [21]). An approach dual to lazy task creation has also been proposed [22] — here the futures are saved for later execution, while the current task continues executing the continuation. Although this technique is easier to implement as it does not explicitly need a support for continuations, it did not evidence a performance comparable to the performance of lazy task creation.

In recent years, as computer systems based on multi-core processors are becoming increasingly ubiquitous, the future as a programming language construct is being resurrected in several modern object-oriented languages. In particular, it is available in Java Platform, Standard Edition API since version 5.0 (recently, a lazy task creation implementation of futures runtime for Java has been described [23]); further, it is included in X10 on a programming language level as a type constructor [9]; even more, there is also a proposal to introduce it in the next version of C++ [13]. In their current version, the .NET platform [4] and in particular C# [3] lack explicit support for futures, though there exists an API providing a support for asynchronous computation.

B. Our Contributions

In this paper, we present architecture overview of a futures runtime for .NET that is based on the lazy task creation. Our

contributions can be summarized as follows:

- We deal with design issues of an efficient runtime for scheduling and executing futures on modern machines with multiple processors that is based on the lazy task creation.
- We introduce the notion of guards as a substitute for future groups [13], [19] with intention to provide a programmer more complex and elegant constructs for programming with futures.
- We briefly sketch a simple extension of the C# language with langauge constructs supporting programming with futures, for which we aim to realize a compiler support.

The rest of the paper is organized as follows. Section 2 gives the rationale behind this work and introduces main constituents of the futures runtime — the concept of futures, the lazy task creation technique and the notion of guards. In Section 3, we present architecture overview of the runtime. The final Section 5 gives concluding remarks.

Acknowledgments.: This work is motivated by an effort to extend the research language AsmL [1] developed at Microsoft Research with new ASM constructs for dealing with the most general type of interaction between algorithm and its environment as introduced in [7], [8]. The whole research was initiated by Yuri Gurevich, to whom we are indebted for clarification of many important issues about the nature of interactive algorithms. Wolfram Schulte pointed us to and discussed relevant related work. Lev Nachmanson helped us to experimentally verify efficiency of an early prototype of the futures runtime on his GLEE graph layout engine [2]. We have also benefited greatly from discussions with Vinod Grover and Daan Leijen.

This work was partially supported by the Phoenix/SSCLI 2006 award from Microsoft Research and Croatian National Science Foundation.

II. The Main Concepts

A. The Rationale Behind

Many programs contain inherent potential concurrency — while there might be parts of the code that must be executed sequentially, the remaining parts may allow concurrent execution. Given a sufficient number of processors, one can achieve the optimal performance by executing concurrently all (sufficiently large) concurrent tasks. In reality, however, the number of available processors is limited, and the optimal performance would be achieved by partitioning the program in a number of appropriately sized tasks such that all processors are busy throughout the program execution.

Since the latter is typically hard (if not impossible) to achieve, the primary motivation for introducing futures and a runtime for their execution is to help a programmer to move at least near to this goal with as little effort as possible. Namely, the idea is to provide the programmer a way to specify what can be safely computed in parallel and then let the runtime to decide how to efficiently carry out the computations. That is, the programmer's only task should be to expose the concurrency in the program by using annotations. However, since the annotated program typically gives rise to tasks of finer grain than the grain of the platform they are executing on, the runtime needs to limit the exploited concurrency to a level allowed by the platform.

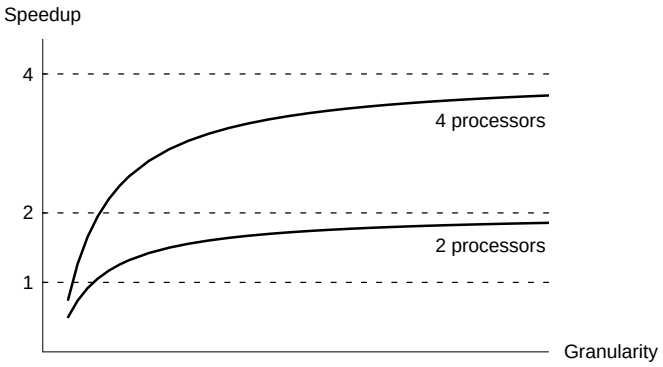

Fig. 1. Performance gain of an abstract runtime with respect to the size of exposed potentially concurrent tasks

The ideal runtime implementation breaks down tasks such that the run-time task granularity is maximized while the system is kept load balanced. Naïve implementations of runtime such as the one based on using a single thread per future or a thread pool do not scale well and are far from the ideal implementation. Not only they lead to significant overhead and degradation in performance, but may even lead to a deadlock. More elaborate techniques, such as lazy task creation are needed in order to achieve a robust and scalable performance.

In general, we can identify three costs related to exposing the potential concurrency and executing the potentially concurrent tasks by the runtime:

1) the cost of *exposing* the potential concurrency,
2) the cost of *creating* a task, and
3) the cost of *switching* between tasks in case of their interdependence.

The cost of exposing the potential concurrency is the key to scalable performance. If this cost is sufficiently small, we can expose the potential concurrency without paying high penalty and facilitate the runtime to take advantage of it, say, when more processors become available in the underlying platform. Obviously, the cost of creating a task should be smaller than the cost of exposing the potential concurrency in order for this strategy to make sense. The cost of switching between tasks is related to the fact that an already scheduled task may not be completed without completing some other task first. In such cases, the runtime must suspend execution of the blocked task and resume execution of another. Although this cost is higher than the other two, its contribution to the total execution time is typically much smaller. The fact that a task might be suspended and resumed later, however, has an impact on costs of exposing and creating tasks.

Figure 1 shows dependence of performance gain of an abstract runtime on the size (e.g. execution time) of exposed

```
public class BinaryTree {
  BinaryTree l;
  BinaryTree r;
  int value;

  public int FutureSum() {
    int s = value;
    future<int> f;
    if (l != null)
      f = l.FutureSum();
    if (r != null)
      s += r.FutureSum();
    if (l != null) {
      wait(f);
      s += f;
    }
    return s;
  }
}
```

Fig. 2. Summation of a binary tree with futures written in the extended C# syntax

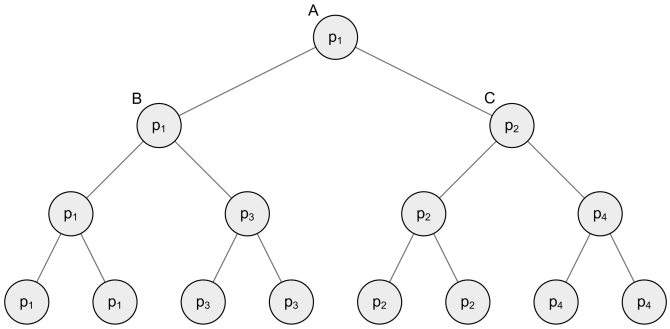

Fig. 3. Execution tree of FutureSum() on 4 processors

potentially concurrent tasks. When exposed tasks are sufficiently large, the runtime achieves the optimal performance, as shown on the right side of the graph. However, more interesting is its left side — it indicates the minimum size of potentially concurrent tasks such that it is sensible to expose them. This cost is proportional to the cost of exposing concurrency. Reducing the cost of exposing concurrency shifts the curve to the left; this way more potentially concurrent tasks can be exposed, which in turn leads to a more scalable performance.

B. Futures and Lazy Task Creation

A *future* is an object that acts as a placeholder for a result of a computation. On a programming language level, the future as a programming language construct is used to denote that some piece of code may be executed in parallel, i.e. to expose potentially concurrent tasks. For instance, in its canonical MultiLisp form, the expression

$$(C\ (\textbf{future}\ F))$$

denotes that a child task computing F may proceed in parallel with its parent continuation C.

As already noted in the introduction, different ways to arrange computation of C and F are possible. In the most straightforward approach, a new task computing F is unconditionally spawned, and the parent task resumes computation of C. In some cases, it is better to inline F and compute it by the current task. The load-based inlining approach [16] makes a separate task computing F only if the system load permits it, otherwise it gets inlined. Our runtime is based on the lazy task creation approach [18] in which F is being evaluated in the current task, while enough information about C is saved so that if some processor becomes idle, C can be moved to a separate task.

In order to illustrate the lazy task creation approach, we consider the example given in Figure 2. It illustrates a simple algorithm (inspired by a similar example given in [18]) summing the nodes of a binary tree. The code is written in our proposed extension of the C# programming language (not formally specified in this paper). It uses a generic type **future**<T> for denoting values that may be computed in parallel. The **wait** construct designates a synchronization point at which the caller will remain suspended until the value of a given future becomes resolved (actually, it represents a *guard* in its most simple form, a notion that we introduce in the next section).

Annotating the variable f in the function FutureSum() as a future indicates that the recursive call to FutureSum() on the left subtree can proceed in parallel with the recursive call on the right subtree. This natural expression of parallelism in FutureSum() gives rise to rather fine-grained concurrency — namely, for a tree of depth k there are 2^k futures.

If FutureSum() would have been run with lazy task creation on 4 processors (say, p_1, p_2, p_3 and p_4), an ideal execution would look as shown on Figure 3. Suppose that a call to FutureSum() on a tree with root A is scheduled on processor p_1. The future at A (representing the call to FutureSum() on B) becomes inlined and executed further by p_1, while its continuation (representing the call to FutureSum() on C) gets stolen by an available processor, say p_2. Likewise, the futures at B and C get inlined, while their continuations are stolen by two remaining available processors p_3 and p_4.

In the described idealistic situation, the lazy task creation maximizes the run-time task granularity and keeps the system ideally balanced. In general, it of course need not be the case, however, experiments (e.g. [18], [23]) show that in non-pathological situations, a user may expect steady and well-behaved performance from this technique. However, we have to note that the cost of maintaining information about the caller's continuation is of great importance for efficient implementation of this technique.

C. Guards

Before using the result of a future, one must make sure that the future is completed and its result is available. There also exist complex situations in which a set of futures must be completed before the computation can be continued. [13] and [19] introduce a construct called *future group* in order to express such conditions. In a typical example, a future group

is used to express that the computation can be resumed if either both futures f_1 and f_2 are completed, or a timeout period has elapsed.

In the context of the behavioral theory of algorithms, [7] postulates the most general form of interaction of an algorithm with its environment. The progress of an interactive algorithm in the sense of [7] depends not only on values of answers provided by the environment but also on the relative timing of when those answers became first available to the algorithm. In the sequel paper [8], a new language construct *guard* is introduced and proved to be sufficient for expressing *any* such dependency.

Guards enable programmers not only to express which events need to happen before further progress of a computation, but also to specify actions to be triggered depending on their relative timings. The semantics of guards uses Kleene's strong three-valued logic — the value of a guard can either be undefined, or it can have a boolean value `true` or `false`. Intuitively, waiting on a guard blocks the computation as long as its value remains undefined. When its value becomes defined, the computation can continue, possibly taking into account the exact value of the guard.

The guards implemented by our runtime have the full expressiveness of guards in [8]. Guards are built from boolean expressions, Kleene's boolean connectives $\curlywedge, \curlyvee$ and $\neg$, and the binary operator on futures $\preceq$. The intended meaning of $f_1 \preceq f_2$ is that f_1 completed its computation before f_2. The role of the three-valued logic can be seen from a simple observation that $f_1 \preceq f_2$ does not have a defined value until at least one of f_1, f_2 is completed, and its value can either be `true` or `false` depending on which one completed first.

More formally, guards are defined by the following inductive definition:

- every boolean expression is a guard,
- if f_1 and f_2 are futures, then $f_1 \preceq f_2$ is a guard,
- if φ and ψ are guards, then $\varphi \curlywedge \psi$, $\varphi \curlyvee \psi$ and $\neg\varphi$ are guards.

The current value of a guard is inductively defined as:

- If e is a boolean expression, then it has its usual boolean value.
- $f_1 \preceq f_2$ has a defined value iff either f_1 or f_2 is completed. If f_2 is not completed or is completed after f_1, then the value is `true`. Otherwise, the value is `false`.
- $\varphi \curlywedge \psi$ has a value `true` if both φ and ψ have a value `true`, and a value `false` if at least one of them has a value `false`. Otherwise, its value is undefined.
- $\varphi \curlyvee \psi$ has a value `false` if both φ and ψ have a value `false`, and a value `true` if at least one of them has a value `true`. Otherwise, its value is undefined.
- $\neg\varphi$ has the opposite value of φ, if φ has a value. Otherwise, its value is undefined.

The most common use of guards is conditioning further progress of the computation on a future to complete. For such purpose, we define $f!$ as a shorthand for $f \preceq f$, which has a value `true` if the future f is completed, or undefined otherwise.

Guards representing commonly used future groups can also be built by Kleene connectives. For example, if f_1 and f_2 are futures, and t is a timeout event, then the guard $(f_1! \curlywedge f_2!) \curlyvee t!$ has a defined value only when both futures are completed or the timeout period has elapsed. A reader familiar with [13], [19] will easily write an appropriate guard representing *any* future group.

III. Architecture Overview of the Runtime

A. Basic Constituents

Threads managed by our runtime are grouped into *thread groups*. A distinct processor in a system is associated with each thread group, and all threads in the same group have their thread affinity fixed to the same processor. The number of thread groups defaults to the number of processors available in the system.

All but at most one thread in a thread group are waiting on a synchronization object (manually reset event) managed by the runtime. The thread not waiting on such event is *running* or *scheduled*, while other threads in the group are *suspended*. If the guard associated with a suspended thread has its value defined, we say the thread is *runnable*.

When a running thread t requires results of other futures in order to continue its computation, it evaluates a guard g expressing the condition for its progress. If the guard has an undefined value, the runtime suspends the thread t and resumes execution of another runnable thread t' in the group as described in the next section. The suspended thread will not be scheduled by the runtime before it becomes runnable again.

Each thread in a thread group has its own task queue — a double ended queue of continuations. When a future is called from a continuation, the continuation gets suspended and placed at the end of the thread's task queue, while the future call is *inlined*, i.e. it gets called immediately.

An integer *depth* is assigned to each continuation at its creation, denoting the number of continuation above in its execution stack. The depth of a thread is the maximal depth of continuations in its task queue.

B. Task Stealing

When a thread t running in a thread group G requires the value of a guard g to be defined in order to continue its computation, it gets suspended by the runtime and the guard g becomes associated with t. After suspending t, the runtime tries to find a runnable thread that can be scheduled in the thread group G.

The runtime first checks if there exists a runnable thread t' in the same thread group. If there exists one, then t' becomes the running thread of the thread group. The exact choice of a runnable thread to be resumed is important, and the runtime always chooses the one with the highest depth in the group. Such choice can be justified by noticing that the chosen thread is probably the one with the most advanced computation and,

thus, resuming it would most likely complete its computation and therefore reduce the total number of threads in the system.

If there are no runnable threads in the thread group G, the runtime traverses thread groups searching for a thread with a stealable continuation. Let t'' be the thread that is being examined at some time instance. The runtime tries to remove the continuation C with the minimal depth from the task queue of t'', and, if successful, starts a fresh thread t' in the thread group G that resumes the stolen continuation. The newly started thread is taken from a thread pool.

The stolen continuation C has originally been put into the task queue of t'' when t'' was assigned a future called from C. When the future called from C gets completed, t'' will try to remove C from its task queue, which is deemed to fail as C is now being executed by t'. Since a continuation is just a closure of a function, t'' cannot proceed before C gets completed, thus t'' will remain suspended on the guard C! until t' finishes execution of C. This way the correct execution order is guaranteed.

IV. Conclusions and Future Work

This paper has presented architecture overview of the futures runtime for .NET that is based on the lazy task creation technique. The work has been inspired by a vast amount of research on this topic done previously for other platforms. To the best of the authors knowledge, the present paper is the first one discussing this topic in the context of .NET platform.

Although the lazy task creation is a well-known technique for efficient execution of futures, we have enriched it with a novel way of hierarchical organization and traversal of tasks in order to reflect the cache/memory hierarchy in modern multi-core processors. Also, our notion of guards is new and allows easy and elegant programming with futures.

Nevertheless, the efficiency and scalability of our implementation needs yet to be determined. Although we have performed some preliminary experimental evaluation on small pieces of code (with an exception of GLEE [2], an application of a moderate size for which we have obtained a speedup of up to 1.7 on a dual-core system by changing only a single for-loop in the code), we leave further analysis of results to a subsequent paper. Also, we plan to perform more tests on larger examples, including some of the benchmarks from the Multithreaded Java Grande Benchmark Suite [20].

References

[1] The AsmL web page. http://research.microsoft.com/foundations/AsmL/.

[2] GLEE web page. http://research.microsoft.com/ levnach/GLEEWebPage.htm.

[3] *ECMA-334: C# Language Specification, 4th edition*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, 2006.

[4] *ECMA-335: Common Language Infrastructure (CLI), 4th edition*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, 2006.

[5] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative task management without manual stack management. In *USENIX Annual Technical Conference, General Track*, pages 289–302, 2002.

[6] Edward H. Bensley, Thomas J. Brando, and Myra Jean Prelle. An execution model for distributed object oriented computation. In *OOPSLA '88: Proceedings of the Conference on Object-oriented Programming Systems, Languages and Applications*, pages 316–322. ACM Press, 1988.

[7] Andreas Blass, Yuri Gurevich, Dean Rosenzweig, and Benjamin Rossman. Interactive small-step algorithms I: Axiomatization. Technical Report MSR-TR-2006-170, Microsoft Research, November 2006. To appear in Logical Methods in Computer Science.

[8] Andreas Blass, Yuri Gurevich, Dean Rosenzweig, and Benjamin Rossman. Interactive small-step algorithms II: Abstract state machines and the characterization theorem. Technical Report MSR-TR-2006-171, Microsoft Research, November 2006. To appear in Logical Methods in Computer Science.

[9] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 519–538. ACM Press, 2005.

[10] Arunodaya Chatterjee. Futures: A mechanism for concurrency among objects. In *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 562–567. ACM Press, 1989.

[11] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI)*, pages 212–223, 1998.

[12] Seth Copen Goldstein, Klaus E. Schauser, and David E. Culler. Lazy threads: Implementing a fast parallel call. *Journal of Parallel and Distributed Computing*, 37(1):5–20, 1996.

[13] Howard E. Hinnant. Multithreading API for C++0X - a layered approach. JTC1/SC22/WG21 - The C++ Standards Committee Document No. N2094=06-0164, September 2006.

[14] Henry C. Baker Jr. and Carl Hewitt. The incremental garbage collection of processes. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*, pages 55–59. ACM Press, 1977.

[15] Robert H. Halstead Jr. MULTILISP: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[16] David A. Kranz, Robert H. Halstead Jr., and Eric Mohr. Mul-T: A high-performance parallel lisp. In *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation (PLDI)*, pages 81–90. ACM Press, 1989.

[17] Barbara Liskov and Liuba Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 260–267. ACM Press, 1988.

[18] Eric Mohr, David A. Kranz, and Robert H. Halstead Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, 1991.

[19] Matti Rintala. Handling multiple concurrent exceptions in C++ using futures. In *Advanced Topics in Exception Handling Techniques*, volume 4119 of *Lecture Notes in Computer Science*, pages 62–80. Springer, 2006.

[20] L. A. Smith and J. M. Bull. A multithreaded Java grande benchmark suite. In *Proceedings of the Third Workshop on Java for High Performance Computing*, June 2001.

[21] Kenjiro Taura, Kunio Tabata, and Akinori Yonezawa. StackThreads/MP: integrating futures into calling standards. In *PPoPP '99: Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 60–71. ACM Press, 1999.

[22] David B. Wagner and Bradley G. Calder. Leapfrogging: a portable technique for implementing efficient futures. In *PPOPP '93: Proceedings of the 4th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 208–217. ACM Press, 1993.

[23] Lingli Zhang, Chandra Krintz, and Sunil Soman. Efficient support of fine-grained futures in Java. In *Proceedings of the 18th International Conference on Parallel and Distributed Computing and Systems (PDCS'06)*, 2006.