

An Enhancement of Futures Runtime in Presence of Cache Memory Hierarchy

Matko Botinčan[‡], Davor Runje[§]

[‡] *Department of Mathematics, University of Zagreb*

Bijenička cesta 30, 10000 Zagreb, Croatia

E-mail: matko.botincan@math.hr

[§] *Faculty of Mechanical Engineering and Naval Architecture, University of Zagreb*

Ivana Lučića 5, 10002 Zagreb, Croatia

E-mail: davor.runje@fsb.hr

Abstract. *A future is a simple abstraction mechanism for exposing potential concurrency in programs. In this paper, we propose an enhancement of our previously developed runtime for scheduling and executing futures based on the lazy task creation technique that aims to reflect the cache memory hierarchy present in modern multi-core and multiprocessor systems.*

Keywords. Futures, lazy task creation, fine-grained concurrency, memory hierarchy.

1. Introduction

The free lunch is over[15, 16] for most sequential and many concurrent applications that are being developed today. New processors coming out on a market do not get their performance boosted by higher clock speeds, yet (typically) by increasingly higher number of cores. Applications that want to utilize the potential performance gain of multi-core processors need to expose the potential concurrency. In order to achieve this, programmers most commonly employ programming techniques based on use of threads and locks, which in turn often lead to error-prone and hard-to-verify implementations. An important challenge of modern software engineering is to invent new and improve existing abstractions that will enable easier and safer dealing with concurrency for the end programmer.

The potential concurrency of many programs is inherently finer-grained than the concurrency of the platform they are execut-

ing on, i.e. the number of processors in the platform is typically much smaller than the number of potentially concurrent tasks in the program. A future [11, 4, 8, 12, 18, 10] 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.

In a previous paper [7], we have presented an architecture of a .NET [3] runtime for scheduling and executing futures that is based on a technique called lazy task creation [13]. The runtime was designed in such way that it allows any function accessible from .NET to be called as a future. Moreover, it also gave rise to a simple extension of C# programming language [2] with constructs for easy and elegant programming with futures.

In modern multi-core/multiprocessor systems, memory accesses are performed through a hierarchy of caches. In order for an optimal performance to be achieved, the runtime should take into consideration the memory organization of the underlying platform. This paper proposes an enhancement of our previously developed futures runtime that aims to reflect this cache memory hierarchy. Its key parts are a hierarchical organization of thread groups and a traversal strategy for task stealing that aims to minimize performance penalties resulting from cache misses and contention between processors.

The rest of the paper is structured as follows. Section 2 explains concepts that play

main roles in the futures runtime — futures, continuations and the lazy task creation. In Section 3, we review the design of the futures runtime and propose its enhancement for systems with hierarchically structured memory. Section 4 gives concluding remarks and discusses some future work.

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

2. Futures and the Lazy Task Creation

A future [11, 4, 8, 12, 18, 10] is an object that acts as a placeholder for a result of a computation. On a programming language level, the future as a 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 [11], the expression

(*C* (**future** *F*))

denotes that the child task computing *F* may proceed in parallel with its parent continuation *C*.¹

The computation of *C* and *F* can be arranged in different ways. Our runtime is based on the lazy task creation technique [13] 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. This way the program is executed sequentially until a concurrent execution becomes possible.

In Figure 1, we repeat the example from [7] in order to exemplify the lazy task creation approach. The example illustrates a simple concurrent algorithm employing futures that sums nodes of a binary tree. The code is written in our proposed extension of the C# programming language. It uses a generic type `future<T>` for denoting values that may be

```
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;
    }
}
```

Figure 1. Summation of a binary tree with futures

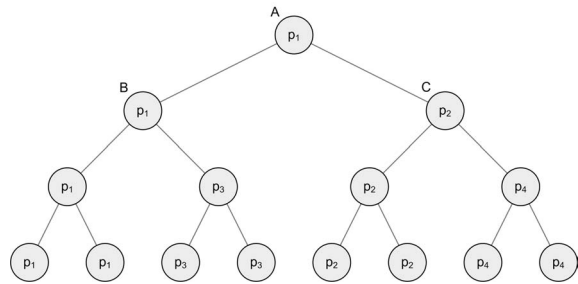


Figure 2. The execution tree of `FutureSum()` on 4 processors

computed in parallel as a future, while the `wait` construct determines a synchronization point at which the caller will remain suspended until the value of a given future becomes resolved.

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 the lazy task creation on 4 processors (say, p_1, p_2, p_3 and p_4), an ideal execution would look as shown in Figure 2. 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*)

¹A continuation is a representation of the execution state of a function performing the computation at a certain point throughout its execution. It is typically represented by the function's stack frame consisting of function arguments, the address of the next instruction and values of local variables.

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 .

The lazy task creation maximizes the runtime task granularity and keeps the system evenly balanced in such idealistic situation. Although, in general, it is not necessary the case, experiments (e.g. [13, 18]) evidence that in non-pathological situations a user may expect steady and well-behaved performance from this technique. Nonetheless, it treats the cost of switching between tasks uniformly, making this approach less likely to perform efficiently in systems with hierarchically structured memory in which cost of switching between tasks depends on their distance in the memory hierarchy.

3. The Futures Runtime in Presence of Cache Memory Hierarchy

The first part of this section reviews design of our .NET runtime for scheduling and executing futures based on the lazy task creation technique from [7]. Afterwards, the second part deals with our proposed enhancement of this runtime for systems with hierarchically structured memory.

3.1. The Design of The Futures Runtime

Before using the result of a future, one must make sure that the future is completed and that its result is available. We employ the concept of guards for conditioning the further progress on a future that is not yet completed. Moreover, 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. Our concept of guards has the full expressiveness of guards from [5, 6]; they are represented as .NET delegates [3].

Threads managed by the runtime are grouped into thread groups. A distinct pro-

cessor 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 a defined value, we say that the thread is *runnable*.²

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 that denotes the number of continuations above in a continuation’s execution stack gets assigned to each continuation at its creation. The depth of a thread is the maximal depth of continuations in its task queue.

When a thread t running in a thread group G requires results of other futures in order to continue its computation, it evaluates a guard g expressing the condition for its progress. Since waiting on a guard blocks the computation as long as its value remains undefined, if the guard has an undefined value, the runtime suspends the thread t . The guard g becomes associated with t , and the runtime tries to find another runnable thread that can be resumed for execution. The suspended thread will not be scheduled by the runtime before it becomes runnable again.

The runtime first checks if there exists a runnable thread t' in the same thread group G . If there exists such, 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.

²One could view thread groups as threads and threads as fibers executed by that thread. If fibers were (still) available in .NET, they could have been used as a natural implementation of this aspect of the runtime.

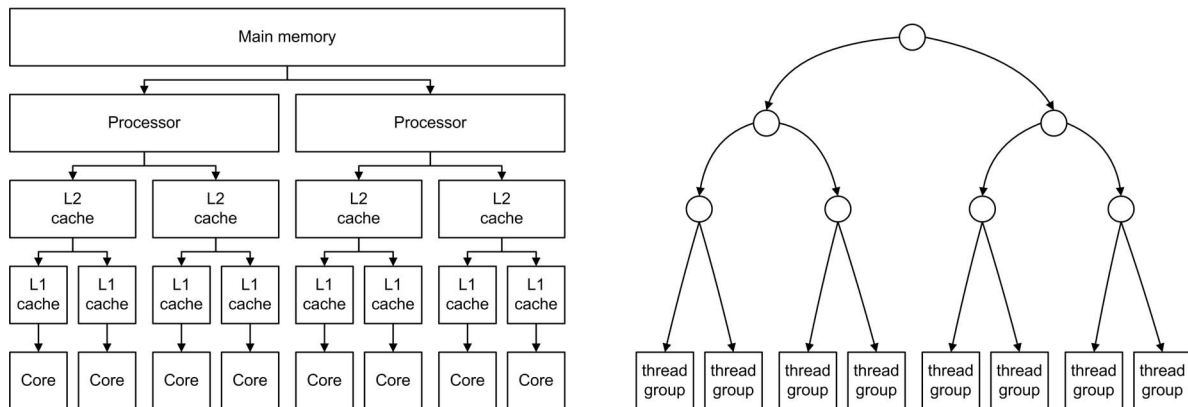


Figure 3. Cache hierarchy and organization of thread groups on a dual Xeon 5300 system

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 G , the runtime traverses thread groups searching for a thread with a stealable continuation. The order that the thread groups are traversed in matters, and the next section deals with an optimal traversal strategy in presence of cache memory hierarchy. Let t'' be the thread being examined at a particular 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 waiting on C until t' finishes execution of C .

3.2. The Proposed Enhancement

In modern multi-core/multiprocessor systems, the shared main memory is accessed

through a hierarchy of cache memories. For example, an Intel quad-core Xeon 5300 processor has four 32-KB Level 1 and two 4-MB Level 2 cache memories, where each Level 2 cache is shared between two cores [1]. In a common configuration, two or more such processors are connected to the shared main memory.

In the original lazy task creation approach [13], processors are examined for a stealable continuation in the round-robin fashion. Such strategy reduces contention on task queues, but ignores the details of memory/-cache hierarchy. A task stealing algorithm should respect the memory organization of the underlying platform in order to achieve optimal performance — stealing a task that has been executed by a core far away in the cache hierarchy results in a performance penalty.

What we propose as an enhancement of our futures runtime along this direction is to employ the binary tree representation of the cache memory hierarchy and use it for hierarchically organizing thread groups with continuations. Here, the shared main memory represents the root of the tree, the processors constitute its leaves, while the shared cache memories are its inner nodes. The corresponding symbolical representation of the system with two quad-core Xeon 5300 processors is shown in Figure 3.

We now formally describe how an optimal strategy for traversing such hierarchically organized thread groups looks like. Let T be the binary tree representation of this hierar-

chy and denote with $L = \{l_1, \dots, l_n\}$ the set of its leaves. A *traversal strategy* t for a tree T assigns a bijection $t_l : \{1, \dots, n\} \rightarrow L$ to every leaf $l \in L$. If for a traversal strategy t and leaves i and j , $t_i(k) = j$ holds, we say that j *occurs* at k -th position in t_i and write $pos_i(j) = k$. We also write

$$t_i : j_1 \rightarrow j_2 \rightarrow \dots \rightarrow j_n$$

to denote that the function t_i maps k to j_k , for $k \in \{1, \dots, n\}$.

An *optimal traversal strategy* for a tree T is a traversal strategy t such that:

- leaves with a smaller distance from i are traversed first in t_i , i.e. it holds:

$$pos_i(j) < pos_i(k) \iff d(i, j) \leq d(i, k);$$

- every leaf i occurs at a position k in some t_j exactly once, i.e. we have:

$$\forall i \in L, \forall k \in \{1, \dots, n\} . \exists ! j \in L \\ \text{such that } pos_j(i) = k.$$

The first condition intuitively minimizes the penalty related to cache misses when moving a continuation to another processor, while the second one intuitively minimizes the contention between processors when concurrently searching for a stealable continuation. Note that an optimal traversal strategy as defined above might not exist for an arbitrary binary tree, however, for every complete binary tree that contains all possible leaves (and the tree representation of the cache memory hierarchy is always such by our assumption) there always exist one.

Example: If we enumerate cores of the symbolical representation in Figure 3 with $0, \dots, 7$ when looking at cores from left to right, then one optimal traversal strategy in the above sense looks as follows:

$$\begin{aligned} t_0 : & 0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \\ t_1 : & 1 \rightarrow 0 \rightarrow 3 \rightarrow 2 \rightarrow 5 \rightarrow 4 \rightarrow 7 \rightarrow 6 \\ t_2 : & 2 \rightarrow 3 \rightarrow 0 \rightarrow 1 \rightarrow 6 \rightarrow 7 \rightarrow 4 \rightarrow 5 \\ t_3 : & 3 \rightarrow 2 \rightarrow 1 \rightarrow 0 \rightarrow 7 \rightarrow 6 \rightarrow 5 \rightarrow 4 \\ t_4 : & 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \\ t_5 : & 5 \rightarrow 4 \rightarrow 7 \rightarrow 6 \rightarrow 1 \rightarrow 0 \rightarrow 3 \rightarrow 2 \\ t_6 : & 6 \rightarrow 7 \rightarrow 4 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 0 \rightarrow 1 \\ t_7 : & 7 \rightarrow 6 \rightarrow 5 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow 0 \end{aligned}$$

Although it is not difficult to calculate optimal traversal strategies as such on the fly, we suggest that in the implementation they are precalculated and stored in a two-dimensional lookup table during the initialization phase of the futures runtime.

4. Conclusions and Future Work

In this paper, we have proposed an enhancement of our previously developed futures runtime for .NET that employs the lazy task creation technique. Although the lazy task creation is a technique known to work well, we have enriched it in a novel way that aims to reflect the cache memory hierarchy in modern multi-core/multiprocessor systems. The key parts are hierarchical organization of thread groups and the notion of an optimal traversal strategy for task stealing that aims to minimize performance penalties resulting from cache misses and contention between processors. The proposed approach, however, has yet to be implemented and its efficiency remains to be determined.

A future is an object that acts like a proxy in the sense of [9]. Even though our proposed extension of C# greatly simplifies exposing the potential concurrency with futures, since C# is a strongly-typed language, parts of the code may need to be manually rewritten in order to use type `Future<T>` instead of `T`, as well as the corresponding `Future<T>`'s values must be explicitly claimed. A possible solution to this problem may be to employ a static analysis based on a qualifier inference for tracking flow of futures through a program and inject appropriate coercions where needed as done in [14].

Furthermore, the observable behavior of a program annotated with futures should be equivalent to the observable behavior of the original program. This property is of course satisfied in case when there are no side effects, however, it is much more difficult (if not impossible) to guarantee it if there exists shared data. Safe futures for Java proposed in [17] enforce semantic safety automatically by using object versioning and task revo-

cation with acceptable performance penalty for programs with modes mutation rates on shared data. We also plan to investigate is it possible, and if yes how, to implement this approach in our settings.

References

- [1] Intel technology and research web page. <http://www.intel.com/technology/>.
- [2] ECMA-334: C# Language Specification, 4th edition. ECMA (European Association for Standardizing Information and Communication Systems); 2006.
- [3] ECMA-335: Common Language Infrastructure (CLI), 4th edition. ECMA (European Association for Standardizing Information and Communication Systems); 2006.
- [4] E. H. Bensley, T J. Brando, M. J. Prella. An execution model for distributed object oriented computation. In OOPSLA '88: Proceedings of the Conference on Object-oriented Programming Systems, Languages and Applications. ACM Press; 1988. p. 316–322.
- [5] A. Blass, Y. Gurevich, D. Rosenzweig, B. 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.
- [6] A. Blass, Y. Gurevich, D. Rosenzweig, B. 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.
- [7] M. Botinčan, D. Runje, A. Vućinović. Futures and the lazy task creation for .NET. In Proceedings of the 15th International Conference on Software, Telecommunications and Computer Networks (SoftCOM 2007). FESB, University of Split; 2007.
- [8] A. Chatterjee. Futures: A mechanism for concurrency among objects. In Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing. ACM Press; 1989. p. 562–567.
- [9] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
- [10] H. E. Hinnant. Multithreading API for C++0X - a layered approach. JTC1/SC22/WG21 - The C++ Standards Committee Document No. N2094=06-0164; September 2006.
- [11] R. H. Halstead Jr. MULTILISP: a language for concurrent symbolic computation. ACM Transactions on Programming Languages and Systems, 1985; 7(4): 501–538.
- [12] B. Liskov, L. Shriram. 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. ACM Press; 1988. p. 260–267.
- [13] E. Mohr, D. A. Kranz, R. H. Halstead Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. IEEE Transactions on Parallel and Distributed Systems, 1991; 2(3): 264–280.
- [14] P. Pratikakis, J. Spacco, M. W. Hicks. Transparent proxies for Java futures. In Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004). ACM Press; 2004. p. 206–223.
- [15] H. Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobbs's Journal, 2005; 30 (3).
- [16] H. Sutter, J. Larus. Software and the Concurrency Revolution. ACM Queue, 2005; 3 (7): 54–62.
- [17] A. Welc, S. Jagannathan, A. L. Hosking. Safe futures for Java. In Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005). ACM Press; 2005. p. 439–453.
- [18] L. Zhang, C. Krintz, S. 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). ACTA Press; 2006.