Karlsruhe Reports in Informatics 2012,9
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

An X10 Compiler for Invasive Architectures

Matthias Braun, Sebastian Buchwald, Manuel Mohr, Andreas Zwinkau

2012

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

Fakultät für **Informatik**

An X10 Compiler for Invasive Architectures

Matthias Braun Sebastian Buchwald Manuel Mohr Andreas Zwinkau
Karlsruhe Institute of Technology
{matthias.braun,buchwald,manuel.mohr,zwinkau}@kit.edu

Abstract

We study the compilation of X10 to novel, highly scalable hardware architectures in the scope of the InvasIC project. To this end, we describe the implementation of a machine code backend and its integration into the existing X10 compiler. In our implementation, the graph-based intermediate representation Firm is used. We identify several issues in the current compiler architecture related to the integration of a low-level backend. The issues and our solutions are independent of Firm and apply to all low-level intermediate languages. Furthermore, we propose optimizations for certain X10 language constructs that are possible on invasive hardware architectures.

Categories and Subject Descriptors D.1.3 [*Software*]: Programming Techniques—Concurrent Programming; C.1.4 [*Computer Systems Organization*]: Processor Architectures—Parallel Architectures

Keywords X10, Machine Code Backend, Invasive Architectures, Resource-aware programming

1. Introduction

Moore's Law of exponentially increasing transistor counts per chip area appears to hold for the next years. However, due to technical issues, chip producers have been unable to translate "more transistors" into "higher clock rates" for the last decade. Instead, multicore architectures were introduced. Extrapolating those trends, we can expect to reach a thousand cores by 2020. Approaches that are currently used do not scale well to that many cores, so hardware architectures will have to adapt. Additionally, we expect future architectures to be more heterogeneous and flexible, providing hardware that can be optimized for different computation scenarios. However, current parallel programming paradigms are not able to deal with the upcoming hardware developments. This prevents the programmer from exploiting all available hardware features. Thus, a paradigm change is needed to let the programmer benefit from future architectures, where hardware properties as heterogeneity and reconfigurability are commonplace.

1.1 Invasive Computing

The Invasive Computing paradigm, developed in the scope of the InvasIC project [11], aims to be an answer to these challenges. It suggests a resource-aware programming model, where the program can dynamically *invade* available resources, e.g., processing units, memory and network connections. An invasion of resources can be restricted by constraints. This allows to request a certain number of resources or to invade dedicated hardware. After the invasion is done, the program *infects* the invaded resources by using them for a certain computation. If the resources are not needed anymore, the program *retreats* from the resources.

Compared to static resource allocation, the dynamic approach improves the overall resource utilization, and hence the efficiency of the system. Furthermore, the invasion of the most suited hardware can improve efficiency. To utilize these improvement possibilities, the hardware must be exposed to the programmer. Thus, the invasive programming paradigm affects the application, the programming language, the compiler, and the operating system.

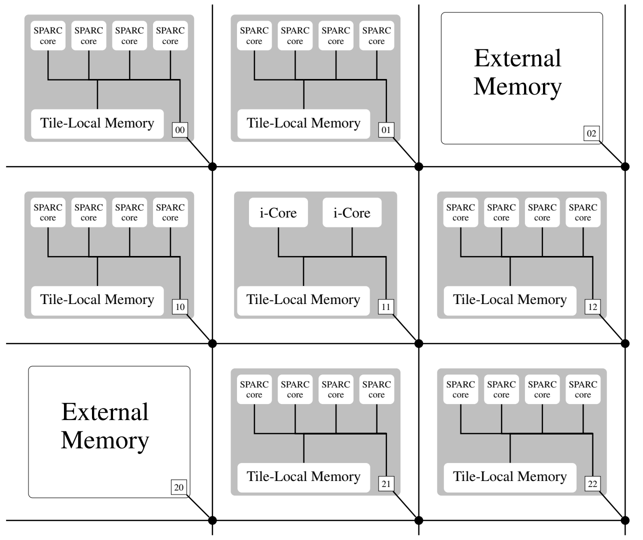

Figure 1. Exemplary invasive architecture: Each tile is connected to the network on chip. Two tiles (02 and 20) are connectors to external memory. The other tiles each contains tile-local memory and either two i-Cores or four SPARC cores.

Figure 1 shows an example of an invasive architecture [9]. It consists of two different variants of SPARC cores. The first variant are standard SPARC cores. The second variant are *i-Core*s, which are reconfigurable SPARC cores that can load accelerators to speed up certain computations [7]. The processors are combined into tiles, which are connected by a network on chip. External memory is attached via special tiles, in addition to per-tile and per-core memory. In contrast to per-core memory, per-tile and external memory are visible in the global address space. Each tile contains a small number of cores (e.g., four), so that cache coherence can be achieved with a bus snooping cache coherence protocol. However, caches in different tiles are not coherent, so the memory architecture scales well with the number of tiles.

As shown in figure Figure 2, an invasive application runs on the invasive runtime support system (iRTSS), which consists of the operating system OctoPOS [8] and an agent system for global resource management. Within the agent system, each application is represented by one agent. If an application wants to invade resources, the application's agent checks whether the request can be fulfilled. In case of multiple concurrent requests for the same resource, the corresponding agents are responsible for finding a

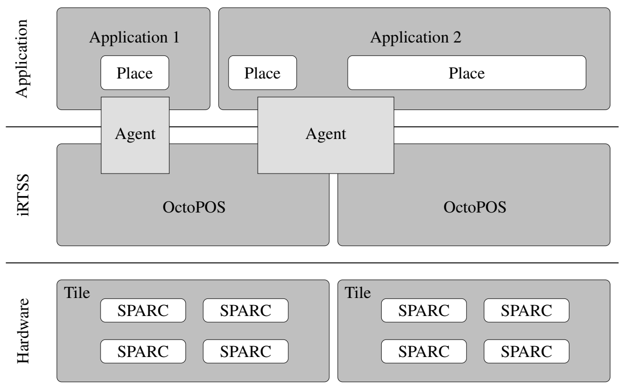

Figure 2. The InvasIC stack: Two tiles of four SPARC cores each are executing two instances of OctoPOS. Two X10 applications are running on top; Each one has an agent for global resource management. Application 2 spans two tiles, so it has two places.

suitable solution. Since only the competing agents are involved, this approach scales well.

On the language level, we encapsulated the resource-aware features in an X10 library. We chose X10 because it already provides various features that make it suitable for programming invasive architectures. For example, the partitioned global address space (PGAS), which models a cluster network, adapts quite naturally to a cache incoherent multi-core architecture. The employable X10 concept is a *place*, where each place belongs to exactly one tile. Thus, threads within the same place have shared memory, whereas threads in different places must communicate with other means.

1.2 Existing X10 Backends

The X10 compiler currently has two backends, called *Managed X10* [10] and *Native X10* [5]. Both backends employ source-to-source translation and target a high-level language. The managed backend outputs Java code and uses a Java compiler to produce Java bytecode. The native backend outputs C++ code and calls a C++ compiler afterward, resulting in a machine code executable.

We decided to create a new backend that targets the intermediate representation of the libFirm compiler. Thus, from libFirm's point of view, the X10 compiler is just an additional frontend. In contrast to the other backends, our backend avoids source-to-source compilation. Instead, libFirm directly generates machine code.

1.3 Firm and libFirm

Firm is a graph-based intermediate representation for compilers. The unique feature of Firm graphs is that they are always in SSA form and hence the SSA form never needs to be deconstructed during code generation. Firm graphs consist of nodes that represent operations and edges that model various dependencies like data dependencies, control flow dependencies and instruction schedule dependencies.

The libFirm infrastructure [1] comprises a collection of frontends, optimizations, backends and tools that all operate on Firm graphs. This includes a C frontend called `cparser` and backends for multiple architectures, with highly-tuned backends for IA-32 and SPARC. The libFirm library has been used for research on code generation, most prominently for modern SSA-based register allocation [2–4, 6].

1.4 Outline

We present our work on a machine code backend for the X10 compiler, which employs libFirm for optimization and code generation. In Section 2 we present the challenges of compiling X10 to machine code as well as our solutions to these problems. Section 3 shows planned adaptations for invasive architectures. Finally, we close in Section 4.

2. Machine Code Backend

As machine code lacks many language mechanisms that are available in high-level programming languages, the architecture of our machine code backend differs in several key points from the architecture of the existing backends. In this section, we show the most important architectural differences, explain why the changes were necessary, present possible solutions and argue why we have chosen a particular solution.

2.1 Strategy for Compiling Generic Classes and Methods

One important feature of modern programming languages is support for generic classes and methods. Java and C++ support this with Java generics and C++ templates, respectively. Therefore, the Java and C++ backends can map X10 generics to the available language mechanisms. However, there is no such mechanism on the level of machine code. This means that, in contrast to the current backends, we have to handle generic classes and methods within the X10 compiler instead of leaving the handling to the post compiler. In the following, we will focus on generic methods but the reasoning for generic classes is the same.

```
public class C {
  public static def id[T](x: T): T = x;
  public static def main(Array[String]) {
    id(42);
    id("Hello World");
  }
}
```

Here, id is a generic method that implements the identity function, i.e., it just returns its argument. It is called twice in the program, once with T = Int and once with T = String. This leads directly to the central question that arises when compiling generic methods: how many versions of id are generated during compilation? Is there one copy of id for each type of argument the generic method is invoked with? This would mean that there is one copy of id that only works for Strings and a separate copy that only works for Ints. Or are there multiple versions that each work for arguments from certain sets of types? Or maybe even a single version that works for arguments of all types?

If only a single version of the method id exists, this version is called *polymorphic* because it works for arguments of multiple types. This approach is used by the Java compiler when compiling to Java bytecode. To enable polymorphism, dynamic dispatch is needed, which, in statically typed object-oriented languages, is usually implemented using the concept of virtual method tables. Furthermore, objects passed to such a polymorphic method must contain additional information to perform the dynamic dispatch. In the case of the virtual method table implementation, the additional information consists of a pointer to the correct virtual method table. Consequently, arguments of primitive types, like integers or floating point values, cannot be directly passed to such a polymorphic function. Java wraps values of primitive types in an object that contains a pointer to a virtual method table, thereby supporting dynamic dispatch. This process is usually called *boxing*.

If a special version of the method is generated for each type, these versions will all be *monomorphic* methods, i.e., they will only work with exactly one type[1]. This is the approach used by C++ compilers to compile C++ templates. Here, one version of the id method is generated for each type of argument. This has the advantage that there is no need for boxing and arguments of primitive types can be directly passed to the appropriate version of the id method. Moreover, type-specific behavior does not have to be expressed through dynamically dispatched method calls. This enables the generation of more efficient code, as the method instantiation for T = Float can use floating point-specific machine instructions.

To compile X10 generics in our machine code backend, we follow the C++ strategy and generate one version of generic methods or classes for each combination of types that is used as an argument. This has substantial performance advantages in the context of X10.

Take, for example, x10.lang.Array[T] and suppose it is instantiated twice in a program, once for T = Double and another time for T = String. The least common supertype of Double and String is x10.lang.Any. The interface Any forms the top of X10's type hierarchy and is implemented by all classes and structs.

Hence, if the compiler generates *one* version of the Array class, the backing array storage holds values of type Any. As Any is just an interface, instances of classes or structs that implement Any can be of arbitrary size and structure. Therefore, the backing array storage can only hold *references* to objects that implement the Any interface. While this is no problem for strings because they are already handled as references to String objects by default, it poses additional overhead for struct types like Double. Their values need to be boxed, and the boxed objects must be allocated on the heap, so that an object reference exists that can be stored in the array.

Thus, in order to turn the availability of X10 structs, i.e., user-defined types with value semantics, into a performance advantage, it is necessary to follow C++'s compilation strategy and generate multiple versions of the same generic class or method.

2.2 Implementation of the Compilation Strategy

The implementation of the aforementioned C++-like compilation strategy for X10 generics requires significant changes to the compilation process. We follow an implicit instantiation approach and instantiate generic classes or methods as needed. This means that generic classes or methods are initially skipped when they are encountered during the traversal of the abstract syntax tree (*AST*). However, as soon as the compiler sees a generic method being called or a generic class being instantiated, it records the method or class and the supplied type arguments and saves this information in a special list. After compilation of all non-generic methods and classes, the compiler generates code for the requested instantiations of generic methods and classes by going over this list. Note that the list can grow while code is being generated if further, previously unseen, instantiations are used.

From an implementation standpoint, the substitution of type parameters with concrete types can either be done explicitly or implicitly. With explicit substitution, the compiler generates a new AST for each instantiation by explicitly substituting the AST nodes of the type parameters with AST nodes that represent concrete types. This approach has the advantage that only the code that generates new ASTs has to deal with type parameters, while the rest of the code always deals with concrete types. The disadvantage is that the compiler needs to build new ASTs for each instantiation, which is a potentially costly operation for big ASTs.

By using implicit substitution, the same AST is traversed multiple times, however, this is done in different contexts. A context in this case is a specific mapping of the type parameters, which are present in the AST, to concrete types. By looking up type parameters in the current type mapping during an AST traversal, the same AST can be used for each instantiation of the generic class or method. This approach has the disadvantage that more code has to be adapted to deal with parametric types, because they are not eliminated in a separate pass before code generation. However, exchanging a context is a much cheaper operation than generating a whole new AST; especially for generic classes, where all non-static methods potentially depend on the type parameter of the class. Therefore, we chose the latter approach and use one AST with different contexts for generating multiple versions of generic methods and classes.

2.3 Generic Native Methods

X10 provides the **native** keyword to mark methods that do not have an implementation written in X10, but are implemented by the runtime of the respective backend or are directly available in the target language. The X10 runtime library often defines methods this way for low-level system integration, where the method is actually implemented in Java or C++. The declaration of a **native** method can be annotated with @Native. The @Native annotations for the existing backends simply specify the code that has to be generated for this particular method at each call site.

Now consider a method that is both generic and native. A typical example is Zero.get[T](), which returns the zero value of the supplied type T.

```
public class Zero {
  @Native("c++", "zeroValue<#T>()")
  public static native def get[T]() {T haszero}: T;
}
```

Due to the @Native annotation, the existing C++ backend translates a call to Zero.get[T]() into a call to a template function `zeroValue<T>()` that is implemented in the C++ runtime. Upon compilation of the C++ code generated by the X10 compiler, the C++ compiler knows which types it has to instantiate `zeroValue<T>()` with and emits a version of `zeroValue<T>()` for each type.

Unfortunately, this approach does not work for a machine code backend. In this scenario, there is no post compiler involved that is able to generate the necessary versions of the generic native functions. Hence, the machine code backend itself must generate the required instantiations. However, native methods, by definition, do not have a method body. Additionally, it is impossible to provide the necessary instantiations in advance, for example as part of a library, because there is no fix set of types the generic native method can be instantiated with.

We solved this problem by handling each generic native method as a special case and hardcoded the semantics of each generic native method in our compiler, essentially making it a built-in function. Typically, the compiler mimics the way the C++ and Java runtimes work and provides multiple implementation variants for different sets of types. In the case of Zero.get[T](), for example, the compiler differentiates between reference types, where **null** is returned, integral types, where 0 is returned, and struct types, where a new object of the specified type is returned after it has been set to zero by a call to `memset`.

To minimize the number of built-in functions, we adapted the X10 standard library to implement as many generic methods as possible directly in X10 rather than declaring them **native**. For example, the generic concatenation operators of x10.lang.String can be implemented in X10 by calling toString() on the argument of generic type, and concatenating the two resulting strings using

[1] In C++, this is not strictly true as instantiations of template functions that take a reference or pointer to T will also accept a reference or pointer to U if U is a subtype of T.

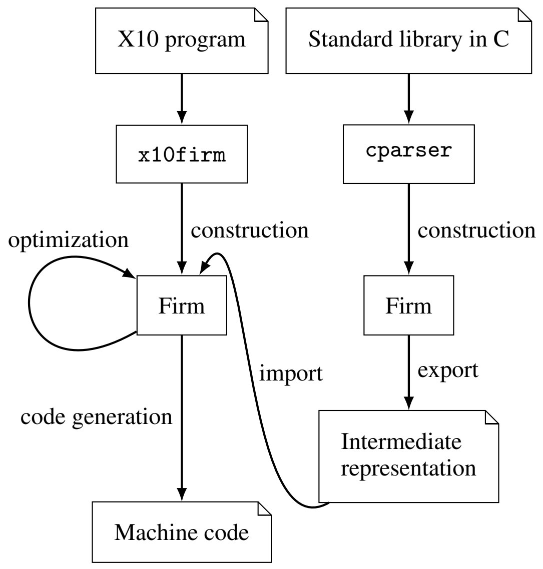

Figure 3. Importing the C part of the standard library into the X10 compilation unit. `cparser` exports its intermediate representation into a file, which is later imported by the X10 compiler `x10firm`. Now interprocedural optimizations process code from both inputs.

a call to the native, but non-generic, concatenation operator that accepts two strings.

2.4 Regular Native Methods

Not only generic native methods but also regular native methods pose a challenge to our machine code backend.

```
public struct Int {
  @Native("c++", "((#0) + (#1))")
  public native operator this + (x:Int): Int;
}
```

As can be seen in this slightly simplified excerpt from the X10 standard library, all basic operations, like adding two integer values, are defined as **native** methods of their respective struct or class type. The existing backends use the aforementioned `@Native` annotation to avoid actually calling a method for operations as simple as an integer addition. In this case, the C++ code generator maps it to the built-in addition of two plain `int` values, which will later result in a single machine instruction.

A machine code backend, however, is again out of luck. While it would be possible to invent a Firm-specific `@Native` syntax for the simple operations and add an additional annotation to each native method, this approach is cumbersome for more complicated methods like `toString()`. Adding each native method as a special case to the our compiler is equally unrealistic because it would require massive engineering effort and would be extremely inflexible. The third option is to implement all native methods in a low-level language like C as regular functions. For the integer addition, the C function just adds its two integer arguments and return the result. On the X10 side, the machine code backend generates a function call to the correct C function. The C files can be compiled separately by a C compiler, resulting in an object file. This object file can later be linked with the output of the X10 compiler. As long as the name mangling conventions of C implementation and X10 compiler are consistent, this approach works.

However, the resulting program is now littered with function calls for operations as simple as an integer addition, which is clearly not desirable. Thus, it is necessary to inline the functions at the individual call sites. This is problematic, though, because the functions are implemented in C whereas the actual program is written in X10. Both parts are compiled separately and only at link time the complete code is available.

Essentially, some form of link time optimization is necessary. This, in turn, means that the internal compiler representation of the C compiler while compiling the C parts of the standard library has to be saved to disk. It can then be loaded by the X10 compiler and used to create one compilation unit out of the X10 program and the C program.

The libFirm API provides serialization and deserialization support for the intermediate representation graphs. As can be seen in Figure 3, it is therefore possible to compile the C part of the standard library with `cparser`, stop before actual code generation and instead export the intermediate representation to a file. The X10 compiler `x10firm` imports this file before compiling the X10 application. It then builds the Firm graphs for the X10 application and the parts of the standard library that are implemented in X10.

At this point, both the C part and the X10 part of the standard library are available to the compiler as Firm graphs at the same time and therefore form one compilation unit. Thus, the regular inlining optimization implemented in libFirm is able to eliminate the call to the integer addition function. Note that the Firm graphs that are inserted at the call site instead of the method call originated from a C program, whereas the Firm graph that contains the calls represents the X10 program. After code generation, the resulting machine code for an integer addition consists of a single instruction.

So far, this approach worked well in our context. It combines the ability to generate efficient code with high flexibility because the C implementations of standard library functions are easily adaptable. We are not aware of any situations where inlining is conceptually not possible, provided that the inlining optimization is powerful enough.

2.5 Status

Our implementation is still in an early state and cannot compile all features of X10. Aside from exceptions, we support the sequential subset of X10. We plan to implement support for the parallel language features within 2012. Since our implementation does not handle the full X10 language, we have not made any performance evaluations yet.

3. Invasive Architecture Support

We use X10 as the primary language in the InvasIC project [11]. As we target a custom operating system that is optimized for the workloads created by typical X10 programs, we plan to change the X10 runtime library and its underlying implementation to make use of the operating system support. Furthermore, there are several opportunities to reduce the communication overhead for **at** expressions in the presence of a global address space.

3.1 Implementing Activities

The X10 execution model is based on work packages, called *activities*, being scheduled on a set of cores in a place. The X10 programming model suggests that activities are lightweight and that creating an activity is a cheap operation. Therefore, typical X10 programs create many activities, some of which are quite small. Hence, X10 activities should not be directly mapped to kernel-level threads, which are relatively heavyweight and costly to create. The existing X10 runtime library employs user-level threads and a work-stealing scheduling policy to reduce the overhead of thread creation and management.

The InvasIC project features the operating system OctoPOS, which is optimized for the workload of typical X10 programs.

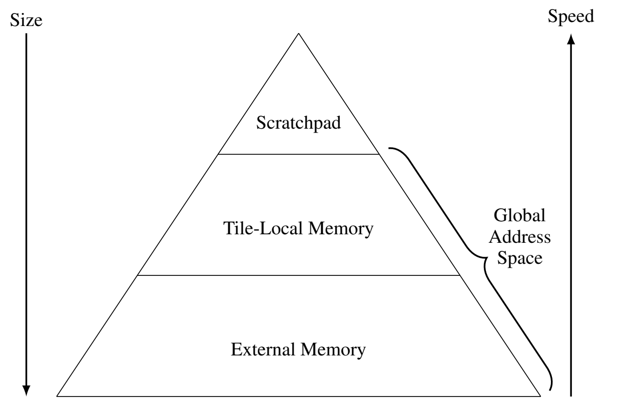

Figure 4. Memory hierarchy of invasive architectures: Memory within the same tile is small and fast, whereas external memory is large and slow. Tile-local and external memory are visible in the global address space.

Instead of heavyweight threads, it offers lightweight *i-Lets* which are tuned for fast creation and context switches. There is no time slicing, although i-Lets are still interrupted for blocking system calls. Therefore, the implementation of the X10 runtime library for OctoPOS will be a thin wrapper around the operating system calls and will map each activity to one i-Let. In particular, there will be no need to implement user-level scheduling.

3.2 Memory Hierarchy

Invasive architectures have a hierarchical memory concept, as illustrated in Figure 4. There is small and fast memory attached to each processor, which is used as cache and scratchpad memory. This is complemented with memory at the tile level. The caches inside a tile are kept coherent through snooping on the tile-local bus.

Finally, all tiles are interconnected by a network on chip, which also connects external memory. This external memory is typically larger than the local memories but also has a higher latency. To achieve high scalability, there is no cache coherence across tiles. Without a cache coherence protocol in hardware, the software has to explicitly flush caches before accessing potentially inconsistent data.

3.3 Programming With Incoherent Caches

Programming with incoherent caches is challenging. Without further knowledge of the cache sizes and algorithms, only the following programming patterns are safe:

- Each memory cell is conceptually owned by a tile. Only the owner is allowed to write to the cell.
- An owner change must be guarded by synchronization constructs.
- A cell may be read by multiple tiles if the owning tile first flushes its write caches, and all reading tiles flush their read caches before reading the cell.

One obviously correct programming model is the partitioning of available memory, assigning a part of it to each tile. This model can be mapped to X10 by creating a place for each tile comprising all memory assigned to it. The exchange of information between tiles only happens on **at** expressions, so the runtime system can ensure a correct sequence of cache flushing and synchronization.

Partitioning the address space inhibits reading a memory cell from multiple tiles, although the architecture would allow this. However, it is possible to take advantage of multiple readers to alleviate the costs of copying data between partitions. We plan a number of changes/enhancements to optimize inter-place communicate for invasive systems.

3.4 Optimizing Inter-Place Communication

The existing X10 implementations use message passing interfaces to exchange data between places. This requires a serialization into packets which are then sent to other places and deserialized there. On an invasive system, we have a global address space and can therefore easily perform random memory accesses without the overhead of preparing data packets. Thus, there is no need for serialization and performing a deep copy suffices.

Note that we expect invasive architectures to enforce a common data layout, so things like endianness and field offsets do not require special attention. We will demonstrate exemplary transformations here, the exact details on when a transformation is possible will be given in the next subsection.

All data used by an **at** expression has to be copied to the target place. Semantically, this copy is performed before the activity created by the **at** is started. However an implementation could place the copy code at the beginning of the new activity without changing any observable program semantics. As it turns out, there are situations where we can delay the copy even further. Consider the following X10 code fragment:

```
val data = /*...*/;
at (someplace) {
  if (abortflag) return;
  process(data);
}
```

This will be transformed to:

```
val data = /*...*/;
at (someplace) {
  val data_copy = deepcopy(data);
  if (abortflag) return;
  process(data_copy);
}
```

We can move the deepcopy right before the first usage of the data, in front of the process call. In the example, this will avoid the copying altogether for the case when abortflag is **true**. We call this *lazy copying*. The transformed code looks like this:

```
val data = /*...*/;
at (someplace) {
  if (abortflag) return;
  val data_copy = deepcopy(data);
  process(data_copy);
}
```

Now suppose the process function does not change any of the data reachable through the data variable. In this case, we can omit the copy altogether. We call this *read sharing*:

```
val data = /*...*/;
at (someplace) {
  if (abortflag) return;
  process(data);
}
```

Often the data transferred between places can be partitioned into independent parts. For example, if the compiler can prove that no common data can be reached by two different variables, then these can be regarded as two partitions, which can be copied independently. Lazy copying and read sharing can then be performed for each partition. We demonstrate this in the following example:

```
val helper_table = /*...*/;
val data = /*...*/;
at (someplace) {
  if (!normalized) {
    normalize(data, helper_table);
  }
  process(data);
}
```

Assuming helper_table and data do not share data, we can optimize them independently. We further assume that helper_table is only read and data is only modified by the normalize function. We can then perform the read sharing optimization on helper_table and for some program paths for data. We can also perform a lazy copying optimization for data. This results in:

```
val helper_table = /*...*/;
val data = /*...*/;
at (someplace) {
  var data_copy = data;
  if (!normalized) {
    data_copy = deepcopy(data);
    normalize(data_copy, helper_table);
  }
  process(data_copy);
}
```

3.5 Lazy Copying and Read Sharing Conditions

In the following, we will determine conditions on when lazy copying and read sharing may be performed safely. Obviously, you cannot delay the copy any longer than the first access to the data. Similarly, using the passed in data reference instead of copying is only legal until the first write operation.

However, in the presence of other activities running on the same place, one might wonder if moving the copy operation is legal at all. Other activities could read or write to the data at the same time and therefore a program with a lazy copy can behave differently than a program with an immediate copy. However, modifying memory that another activity is reading before any synchronization has been performed is a data race. As data races trigger undefined behavior anyway, a lazy copy is legal behavior. So before the first explicit synchronization operation, we do not need to worry. The memory copy must have been performed before reaching a synchronization operation.

This leads to the following rules. Considering an **at** expression with a data partition D, the following conditions have to hold for each path from the beginning of the **at** expression to any program point inside it:

- A copy must be present before any write operation to data in D.
- A copy of D must be present before any synchronization operation.
- Before a copy has been performed, read operations to data in D may simply access the data through the passed reference.

A program analysis can be performed for each data partition. The analysis must conservatively approximate the point with the first read, write or synchronization. If a reference to the data cannot be tracked at some point, then any code accessing unknown values must be considered as reading and writing. Hence, a copy has to be performed before.

4. Conclusions and Future Work

We have demonstrated how a machine code backend can be integrated into the current X10 compiler. Furthermore, we have shown how properties of invasive architectures can be used by the compiler to optimize X10 programs. As our current implementation is in an early stage of development, our next step is to improve the machine code backend to support the full X10 language. Finally, we will evaluate the X10 programming model and possible optimizations in the context of invasive architectures.

5. Acknowledgments

This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89). Thanks to Eduard Frank and Julian Oppermann for implementing large parts of the X10-Firm integration.

References

[1] M. Braun, S. Buchwald, and A. Zwinkau. Firm—a graph-based intermediate representation. Technical Report 35, Karlsruhe Institute of Technology, Dec. 2011.

[2] M. Braun and S. Hack. Register spilling and live-range splitting for SSA-form programs. In *Proceedings of the International Conference on Compiler Construction*, pages 174–189. Springer, Mar. 2009.

[3] M. Braun, C. Mallon, and S. Hack. Preference-guided register assignment. In *International Conference on Compiler Construction*. Springer, Mar. 2010.

[4] S. Buchwald, A. Zwinkau, and T. Bersch. SSA-based register allocation with PBQP. In J. Knoop, editor, *Compiler Construction*, volume 6601 of *Lecture Notes in Computer Science*, chapter 4, pages 42–61. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2011.

[5] D. Grove, O. Tardieu, D. Cunningham, B. Herta, I. Peshansky, and V. Saraswat. A performance model for X10 applications. In *ACM SIGPLAN 2011 X10 Workshop*, June 2011.

[6] S. Hack, D. Grund, and G. Goos. Register allocation for programs in SSA-form. In A. Zeller and A. Mycroft, editors, *Compiler Construction 2006*, volume 3923 of *Lecture Notes In Computer Science*, pages 247–262. Springer, Mar. 2006.

[7] J. Henkel, L. Bauer, M. Hübner, and A. Grudnitsky. i-Core: A run-time adaptive processor for embedded multi-core systems. In *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA 2011)*, July 2011. invited paper.

[8] B. Oechslein, J. Schedel, J. Kleinöder, L. Bauer, J. Henkel, D. Lohmann, and W. Schröder-Preikschat. OctoPOS: A parallel operating system for invasive computing. In R. McIlroy, J. Sventek, T. Harris, and T. Roscoe, editors, *Proceedings of the International Workshop on Systems for Future Multi-Core Architectures (SFMA)*, volume USB Proceedings of *Sixth International ACM/EuroSys European Conference on Computer Systems (EuroSys)*, pages 9–14. EuroSys, Apr. 2011.

[9] R. K. Pujari, T. Wild, A. Herkersdorf, B. Vogel, and J. Henkel. Hardware assisted thread assignment for RISC based MPSoCs in invasive computing. In *Proceedings of the 13th International Symposium on Integrated Circuits (ISIC)*, Dec. 2011.

[10] M. Takeuchi, Y. Makino, K. Kawachiya, H. Horii, T. Suzumura, T. Suganuma, and T. Onodera. Compiling X10 to Java. In *ACM SIGPLAN 2011 X10 Workshop*, June 2011.

[11] J. Teich, J. Henkel, A. Herkersdorf, D. Schmitt-Landsiedel, W. Schröder-Preikschat, and G. Snelting. Invasive computing: An overview. In M. Hübner and J. Becker, editors, *Multiprocessor System-on-Chip – Hardware Design and Tool Integration*, pages 241–268. Springer, Berlin, Heidelberg, 2011.