On Implementing High Level Concurrency in Java

G Stewart Itzstein and Mark Jasiunas

School of Computer and Information Systems
University of South Australia,
Adelaide, South Australia 5095,
Email: itzstein@cs.unisa.edu.au

Abstract. Increasingly threading has become an important architectural component of programming languages to support parallel programming. Previously we have proposed an elegant language extension to express concurrency and synchronization. This language called Join Java has all the expressiveness of Object Oriented languages whilst offering the added benefit of superior synchronization and concurrency semantics. Join Java incorporates asynchronous method calls and message passing. Synchronisation is expressed by a conjunction of method calls that execute associated code only when all parts of the condition are satisfied. A prototype of the Join Java language extension has been implemented using a fully functional Java compiler allowing us to illustrate how the extension preserves Join semantics within the Java language. This paper reviews the issues surrounding the addition of Join calculus constructs to an Object Oriented language and our implementation with Java. We describe how, whilst the Join calculus is non-deterministic, a form of determinism can and should be specified in Join Java. We explain the need for a sophisticated yet fast pattern matcher to be present to support the Join Java compiler. We also give reasons why inheritance of Join patterns is restricted in our initial implementation.

1 Introduction

Java has made concurrent programming using threads widely available to mainstream programmers. However, this situation has just re-emphasised what many experienced concurrent programmers already knew, that concurrent programming is inherently difficult. It is easy to make a mistake in a complex application that uses low-level synchronisation constructs such as monitors [15]. Object Oriented designs doesn't necessarily make concurrent programming easier. A poorly designed concurrent object oriented program can easily obscure the behaviour of threads running in parallel. Unlike processes in operating systems, which are protected by memory management software (other than those explicitly given all privileges), Java uses a type system to protect users from executing unsafe operations. However, the Java type system does not protect the user from concurrent access to shared variables. For example, programming a thread pool using only monitors can be a non-trivial task. The programmer needs to pass references

to (usually objects implementing the runnable interface) a job dispatcher. This job dispatcher via some central registry of workers finds which threads are idle and signals a thread that a job is waiting. The worker thread then collects the job and runs it returning the outcome via some shared variable. Implementations of this pattern can be quite difficult with shared data being vulnerable to corruption due to double updates.

With the increasing interest in concurrent applications such as enterprise information systems, distributed programs and parallel computation there seems to be a need to provide a higher-level abstraction for concurrency and synchronisation. This would provide the ability to directly represent much higher-level abstractions in concurrent programs.

The Join Java compiler generates standard byte-code allowing compiled code to run on any Java platform. The core compiler used for Join Java compiler was based upon the extensible compiler developed by Zenger [24]. Thread synchronisation in Join Java is as straightforward as writing and calling standard Java methods. We have previously shown [17, 16] how Join Java incorporates concurrency semantics from the formal Join calculus [9] and that this may allow a more rigorous investigation of the behaviour of implementations, potentially reducing subtle errors.

It could be argued that many of the advantages claimed above can be achieved by using a pre-compiled library of high-level concurrency classes. In fact a number of approaches have been taken in this direction [13, 23], a lot of which follow the approach of Hoares [14] Communicating Sequential Processes. However extending the language rather than supplying a library allows the compiler to better utilise resources to support concurrency, as the synchronisation mechanism is an integral part of the language. For example, mandatory use of libraries are difficult to enforce, a programmer may choose to use one feature but not another (or even forget to call a method at the correct time) which leads to potential undiagnosed problems at runtime. Further information about these problems can be found in [4]. If the language is implemented as a pre-processor any syntax errors are related to the programmer in terms of the output of the pre-processor not the source file that the programmer is familiar with.

In this paper when we refer to Java we mean the language itself and not necessarily the many libraries (such as the API [12] or Triveni [5]) that have been added to support particular application domains.

The paper begins by providing a motivation and background for the language extension. Section two provides a brief overview of Join Java. In section three we look at some of the language semantic issues we encountered implementing Join semantics into the Java language. Section four examines the pattern matcher that the Join Java extension uses to resolve method calls at runtime. Section five draws some conclusions and briefly examines possible future work.

1.1 Motivation

Why introduce yet another language extension into Java? More and more programmers have to deal with problems with both concurrency and the related

synchronization. Most modern production languages supply support for concurrency and synchronization using language technology that is nearly 30 years old [15]. The reasons for this are two fold. Firstly language developers try to make the language as expressive as possible by implementing low-level language constructs. If the programmer requires high-level concurrency semantics they are assumed to be able to implement these as a library. Providing a higher-level concurrency semantics allow the programmer to *choose* to use either the high-level construct or the low-level construct. In this way we support the programmer choosing an appropriate balance between performance and safety. A similar implementation to Join Java called Polyphonic C# has been recently announced [2]. We however believe that Join Java has more straightforward syntax and semantics than the Polyphonic C# proposal. Our implementations syntax restricts the expression of different behaviours to a single place in the Join pattern. We have also introduced an **ordered** modifier that provides a simple priority ordering to declarations. This gives the programmer choice in the type of determinism this way simplifying the declaration of some problems. We examine this more closely later in the paper.

1.2 Join Calculus

In this section we give an overview of the Join calculus [9] and introduce some of the terminology we have adopted. Join calculus can be regarded as a functional language with Join patterns. The Join Java extension semantics are based on the Join algebra originally proposed by Fournet. This calculus can be thought of as both a name passing calculus (i.e. processes and channels have identifiers) and a core language for concurrent and distributed programming [18]. Traditionally Join operational semantics are specified as a reflexive chemical abstract machine (CHAM) [3, 18]. Using this semantic the state of the system is represented as a "chemical soup" that consists of active definitions and running processes [18]. Potential reactions are defined by a set of reduction rules. When the soup contains all the terms on the left hand side of a reduction rule the terms react and generate all the terms on the right hand side of the reduction rule.

In Join Java we call the individual terms on the left hand side of the reduction rule, *Join fragments*. We call the entire left-hand side of the reduction rule a *Join pattern* and the entire rule a *Join method*. When all the fragments required to fulfil a Join pattern exist in the soup (in our case a pattern matcher object) the body of the Join method is executed.

Join patterns can also be viewed as guards on the message passing channel. When all the fragments of the Join pattern are called the "guarded" message (in Join Java these are the parameters of the call) is transferred between callers of the Join fragments. The standard Join calculus does not support synchronous Join fragments however, a formal translation based on CPS [22, 1] is available from [9, 18] INRIA to convert expressions containing synchronous names to asynchronous fragments. Join calculus patterns can thus be mapped directly to Join Java patterns.

```
final class SimpleJoinPattern {
   //will return value of x to caller of A
   int A() & B() & C(int x) { return x; }
}
```

Fig. 1. Join Method Declaration

2 Join Java

In this section we will introduce the syntax of our super-set of Java. Join Java makes a number of syntactic additions to Java. The main two being the addition of Join patterns and the addition of an asynchronous return type signal.

A Join method (see figure 1) in Join Java gives the guarded process semantics of the Join calculus to Java. That is the body of a Join method will not be executed until all the fragments of a Join pattern are called. If Join patterns are defined with pure Java return types such as **void** or **int** they have blocking semantics. If the return type of the leading fragment is the new type **signal** the method header is asynchronous (an early return type). Trailing Join fragments are always asynchronous in the current version of the language, that is they will not block the caller. A non-Join Java aware class can call methods in a Join Java class even if the return type is **signal**. In the case of a **signal** return type the caller will return immediately. In figure 1 we can see a example of a Join method declaration within a Join Java class.

In Figure 1 the method would be executed when calls are made to all three methods (*A() B()* and *C(int)*). A call to method *A()* will block the caller at the method call until methods *B()* and *C(int)* are called due to the requirement that a value be returned of type *int*. When all method fragments have been called the body of the corresponding Join method is executed returning the *int* value to the caller of *A()*. The message passing channel in the example is therefore from the caller of *C(int)* to the caller of *A()* as there is an integer value passed from the argument of *C(int)* to the return type of *A()*. The call to *B()* only acts as a condition on the timing of the message passing. One thing to note is that the fragments A, B and C do not have method bodies of their own. The invocation of a fragment does not invoke any method body. Only when a complete set of fragments that form a Join pattern have been called does a body execute. The main advantage of the Join patterns in the language is message passing. In figure 1 we see that a simple unidirectional communication channel exists between the caller of *C(int x)* and *A()*. The call to *A()* will be blocked until a call to *C(int x)* and *B()* exists. When that occurs the argument x is passed from the caller of *C(int x)* to the caller of *A()*.

A further change to Java that relates to the introduction of Join patterns is the addition of two class modifiers **ordered** and **unordered**. These modifiers alter the behaviour of the pattern matcher for the Join patterns of the current class. Firstly, **unordered** (the default behaviour for a Join Java class) exhibits random behaviour that simulates non-determinism when confronted with multiple possible matches. For example figure 2 shows a Join Java class in which we have two transitions and a constructor. When an object of type UnorderedExam-

```
final class UnorderedExample {
   //Constructor
   UnorderedExample() { B(); A(); S(); }
   signal A() & S() { System.out.println("Transition1"); }
   signal B() & S() { System.out.println("Transition2"); }
}
```

Fig. 2. Join Method Declaration

```
final class SimpleJoinThread {
   /*code that uses x*/
   signal athread(int x) { }
}
```

Fig. 3. Join Method Declaration

ple is created, Join fragments *A()* and *B()* are called followed by fragment *S()*. The pattern matcher has a non-deterministic choice to make. Both transitions one and two can be matched but not both. With the **unordered** modifier the pattern matcher will make a random determination which method to complete. However if the modifier was changed to **ordered** the pattern matcher will give precedence to the first pattern that is defined in the class (hence the designation **ordered**), in this case transition one.

The final major change to Java is the introduction of a **signal** return type indicating that the method is asynchronous. Any method with a **signal** return type specifies that on being called a thread will be created and started. In reality the compiler optimizes this thread creation to only create threads when necessary. For instance only for a Join method with a **signal** return type for the leading fragment will a thread will be created; in all other cases no thread will be created. This is often a convenient way of creating independent threads of execution without having to create subclasses of threads with shared variables to pass arguments in. Figure 3 shows an example declaration of a thread with argument x.

2.1 Language Syntax

The first issue we examined was how do we express the Join calculus in Join Java. Our primary requirement was that we had to try and express the Join calculus so that it would be intuitive to the user. For this we looked at a number of syntactic variants [7, 8] of the Join calculus to find one that seemed to be sympathetic to the Java language syntax. We eventually settled on a syntax similar to that proposed by Odersky that was later implemented in the Funnel language [20, 21]. It is worth noting that the syntax concurrently developed by Benton at Cambridge [2] for C# has a number of similarities to our language. However, there are two main differences. The Polyphonic C# language allows the synchronous method (in Polyphonic C# a Chord) to be any one of the fragments where in Join Java we restrict this to being the first fragment only. Whilst this allows flexibility in the writing of the methods we felt that by locking the synchronous/asynchronous choice to the first fragment we make it more

obvious to the programmer or code reader what the synchronization behaviour of the Join Pattern is. Of course in Polyphonic C# the author can simply reorder the method to place the synchronous method first however, this freedom would lead to more unreadable code. Secondly Polyphonic C# does not support the specification of the deterministic policy for resolving ambiguous reductions of the Join Patterns. We have implemented the **ordered** and **unordered** keywords because we believe the programmer may want more control over the evaluation of Join patterns. In this way the programmer will have control of the policy for evaluating ambiguous reductions. Future work mentions possible improvements to the determinism modifier. For a more complete coverage of the Join Java language see [17] and [16].

3 Language Semantic Issues

In this section we are going to look at the three main semantic issues that have arisen during the implementation of Join Java. These issues are firstly how we handle the possibly detrimental combination of inheritance and high-level concurrency. Secondly how do we handle the non-determinism in the Join calculus in which the high-level concurrency and synchronization are based upon? Finally if we are implementing Join in our language how do we support the *return-to* construct that the calculus supplies when the Java language does not support multiple return values?

Inheritance in Join Java is supported by the standard Java language. However, we do not allow Join patterns to be inherited. The reason for this is that it has been observed that the semantics of inheriting Join patterns [11] in an earlier non-mainstream Object Oriented language led to subtle behavioural differences. The main one of which is the inheritance anomaly initially described by Matsuoka [19]. This is likely to lead to more error prone and unexpected side effects unintended by the programmer. It could be suggested that this makes the language non-object oriented. However, there are a number of examples where Object Oriented languages introduce features into the language where these features either restrict or change the Object Oriented nature of the language. For example static modifiers change the nature of the language from that of Object Oriented data structure to something akin to procedural data structures. In the future we are going to pursue the interaction of the inheritance anomaly with Join Java. We note that our synchronization mechanism is at the method level and hence should reduce the possibilities of inheritance anomalies appearing [6]. We also note that inheritance is omitted in the polyphonic C# proposal for presumably the same reasons.

Another difference between Join Java and earlier non-object oriented Join calculus implementations is the single return structure. That is having blocking semantics (**void**, **int** etc...) in two or more fragments of a Join pattern. There are two reasons why we made the restriction to the current version of the language. Firstly the Java language does not have a cognitive construct for multiple return paths from a single method, that is to say there is no **return-to** construct. This

means that programmers of the Java language would have difficulty in connecting this idea to the language. The second issue we observed with implementing the **return-to** construct is the complexity (and hence the speed) of the pattern matcher increases significantly due to the overhead of tracking multiple localities of method call origins. We have found that by paying the small penalty of extra Join patterns you can implement bi-directional channels as two single direction channels anyway.

During the implementation of benchmark examples for the compiler we found that the non-deterministic nature of the language made solving certain problems more difficult (see [16] for a state machine and Petri-net example). This deficiency was solved by the introduction of determinism to the language via the class scoped **ordered** modifier. Simply put when a Join class has the **ordered** modifier switched on and a situation occurs when two or more Join patterns could execute preference is given to the first one defined in the class. If the **ordered** modifier is not switched on the pattern matcher will make a pseudo-random choice of which pattern to execute.

4 Pattern Matching

A major component of the Join Java extension is the pattern matcher that is used at runtime to decide which Join calls are matched together to execute Join method bodies. A prototype pattern matcher has been implemented in the form of a runtime library that integrates closely to the compiled code from the Join Java compiler. The pattern matcher implements dynamic channel formation as defined in the CHAM [10] operational semantics. The matcher forms the core of the runtime system. For every class containing a Join pattern a separate pattern matcher is generated. Each time a method is executed in one of these Join-enabled classes the signature and arguments of the call is sent to the pattern matcher for processing. The pattern matcher will determine if the Join fragment along with any previous calls completes a Join method. In the case where it does not complete a Join pattern the pattern matcher will queue the new fragment. If the fragment is of asynchronous type, control is returned to the caller. Otherwise the call is a synchronous style call and the caller is made to wait until the fragment is used in a complete pattern. This section details the pattern matcher component of the runtime support library. It describes the operation and importance of this part of the system. The pattern matcher needs to be very fast and memory efficient. We mention a number of previous designs such as state based and bitmap implementations, and finally we talk about our prototype tree matcher. Pattern matcher policies are introduced and the relative merits of each type of matching policy are elaborated. We give reasons for our choice of policy and how that affects the predictability of programs.

The pattern matcher takes as its input a stream of requests to form channels in the form of calls to Join fragments. Each fragment provides only partial information for the construction of a channel. It is the roll of the pattern matcher to monitor calls to the Join fragments. When there are enough fragments to

complete a Join method the pattern is said to be completed and the arguments for all the fragments are passed to the body of the Join pattern along with the return location. The pattern matcher itself does not perform any operations on the data it simply maintains references to all waiting fragments and the origin of the calls of any synchronous fragments. The pattern matcher can be viewed as a type of scheduler when more than one pattern is potentially completed after a Join fragment call. In this case the pattern matcher requires a policy on determining what pattern to complete. Another way of viewing the pattern matcher is a mechanism for marshalling the parameters required for the body of each Join pattern. Each call to a Join fragment has a unique set of real parameters that are stored and then forwarded when ready.

When the pattern matcher is asked to process a new Join fragment there may be two or more possible patterns that can be completed by the fragment. For example given two Join patterns **A&C** and **B&C**, when the program is run an **A** and a **B** is called followed by a **C**. In the Join calculus there is non-deterministic choice as to what pattern should be chosen. The pattern matcher should be guided by a policy in this case. An example policy might be longer Join patterns are completed first. This policy could result in starvation of shorter patterns in some cases. The implementer of the pattern matcher must be careful that the algorithm does not introduce unexpected biases into the pattern matcher. Any choice other than the purely pseudo non-deterministic (random) policy should be predictable to the programmer.

The pattern matcher is implicitly a searching problem. That is when a new fragment arrives it has to be checked to see if there are any completed patterns. It will therefore always take some finite time to find a match. The Join Java pattern matcher needs to maintain state information about the callers of the blocking and non-blocking methods. This makes the search problem somewhat unusual requiring an especially constructed solution. The speed of the search is dependant on the number of patterns and fragments in each pattern as well as the algorithm chosen and the prevailing policy. Of course some optimizations are possible in the search process to reduce execution time. However, each different search algorithm has some limitation that makes it non-ideal for some set of problems. In all cases we should aim to have the pattern matcher whose performance degrades predictably as the size of the Join Java program increases. As we will explain later usually implies a limit on the number and size of patterns that can be registered.

A further design decision is whether to invoke the pattern matcher as a library call to a runtime system thereby doubling the method call overhead and argument passing. An alternative is to extend the JVM with new byte codes so that the pattern matcher is implemented as an extended virtual machine. The advantage to the latter is that the matching code is implemented on the native level. This would increase the performance at the expense of portability. A native method call for the pattern matcher is also an option however, this would of course be the slowest approach with no discernible advantage as the

boundary between the native environment and the Java environment is slow to negotiate. In our prototype we have chosen to use the library call approach.

There are two ways of implementing the pattern matcher either by using a threaded and unthreaded model. When a fragment arrives at the pattern matcher, should it take over the call for processing with its own runtime or should it borrow some of the callers' runtime. By borrowing some of the callers' runtime we simplify the design of the matcher. We just have to lock the mutator and accessor methods of the pattern matcher whenever there is a change to the state of the data structure so that the integrity of the pattern matcher is maintained. In our implementation we have chosen the unthreaded model for simplicity.

4.1 Compiler Interface to the Pattern Matcher

The compiled version of the Join Java program will create an instance of the pattern matcher for each class that contains Join or asynchronous methods. This is done the moment the first call to a fragment is made. The compiler generates *join.system.joinPatterns alllocal = new join.system.joinPatterns(this);* to create an instance of the pattern matcher. The compiler adds a new Join pattern to the pattern matcher data structure with the method call *alllocal.addPattern(new int[]2, 3, true);* The second segment of code shows how the definition of a Join Pattern is passed to the pattern matcher. The first argument is an array of **int** that shows the Join fragment ids that take place in the pattern. These are in the order they are defined in the class. The second argument is a boolean that defines whether the pattern is a synchronous (true) or asynchronous (false) Join pattern. When this *addPattern* method call is made the Join pattern is added to the data structure and the individual Join fragments are linked to all the existing patterns that reference the fragment in the data structure including the one that has just been added. Join pattern addition is only done once in the runtime life of the Join class and requires a traversal of the pattern matcher data structure which modifies the data structure accordingly.

When a Join fragment is called the pattern matcher is invoked with the *addSyncCall()* for synchronous fragments and *addCall()* for asynchronous fragments. This method searches the data structure for a completed Join pattern. There are two possibilities for any identified Join pattern that is involved with the Join fragment. First if the other Join fragments in the referenced pattern have not been called yet the pattern does not complete. In this case the instance of the call is registered in the data structure and the pattern matcher method call returns and blocks the fragment caller if it is synchronous otherwise the method simply returns asynchronously. The second possibility is that one or more patterns are completed by the Join fragment that was just passed to the pattern matcher. In this case the policy settings of the pattern matcher may need to be checked to select which of the multiple patterns that the call has completed will be returned for execution. Of course there is no policy needed if only one pattern is matched. The pattern that is selected to fire is then returned to the callee along with the fragments (with their arguments) and those fragments

are removed from the data structure. It is possible and sometimes likely that there will be multiple instances of each Join fragment waiting for completion. It is advisable that these fragments be removed in FIFO order due to fairness. Once the Join fragments that have been selected to cause the completion of the pattern have been identified by the policy; the reference to these are passed back to the callee. This allows the thread associated with the fragments to be unblocked by notify calls. The following code is executed when an synchronous Join fragment is called. *join.system.returnStruct retval= all.addSynchCall(new java.lang.Object[] new java.lang.Integer(par1), 0, this);* The return structure contains the arguments for the pattern (collected from the various fragments) and the completion status. If the completion status is **true** the return structure is passed to a dispatch method in the Join class that executes the appropriate Join pattern body. If the completion status is **false** the caller is blocked until the completion status becomes **true**. The *addCall* (for async return types) acts in a similar way except the caller is not blocked.

4.2 Pattern Searching

There are three major facets of the implementation of the pattern matcher. First there is the data structure used in the pattern matcher to store the status of the pattern matching pool, next there are the algorithms that traverse and search for elements in this data structure and finally there are the policies that resolve non-deterministic/deterministic situations in the pattern matching process. Most of the discussion in the remainder of this section concentrates on the techniques for pattern matching we designed. Firstly we briefly examine previous approaches to pattern matching in this application domain. We then examine how we implemented algorithms for searching for completed patterns in our runtime system. Finally we cover a potential optimization that could be used for our system in order to overcome problems with state space explosion and runtime delays.

4.3 Previous Approaches to Pattern Matching

There are a number of approaches that have been taken in constructing a pattern matcher to achieve the semantics of the Join calculus. The simple approach is to design the pattern matcher to record the state of the pool. We then record all possible Join patterns (reductions) in a list. These possible patterns are then linearly compared against the current state of the pattern matcher. If there is a match the pool is updated and the result is returned. However, this approach breaks down, as search is expensive on every call to the pattern matcher. A second more sophisticated approach to pattern matching is that used by the original Join language [7] used a state machine to represent all possible states the pool could be in. However the designers found that state space explosion was a problem and they used state space pruning and heuristics to attempt to reduce the problem. The second version of their language used a bit field to represent the status of the pattern matching. Each pattern reduction was compared with the current state of the calls via an XOR call atomically. Whilst this approach

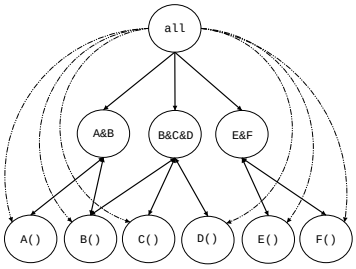

Fig. 4. Internal Representation of Tree Pattern Matcher

sped up pattern matching enormously the solution is not scalable beyond the predefined maximum size of the bit-field. The state space implementation consumed a lot of memory and the bit field solution was limited on the upper end by the max number of digits and hence Join fragments that the Join patterns could have.

4.4 Tree Based Pattern Matcher

In our pattern matcher we have tried to find a middle ground between the space complexity of a state-based solution and the time complexity of a linear solution. We have achieved this by using a tree structure to represent the static structure of the patterns of a Join class. The idea of our approach is to limit the search space during the runtime of the program. We therefore designed the data-structure with the idea of a localized searching in mind. In our data-structure interior nodes represent patterns and leaves represent Join fragments. The root acts as an index to both leaves and interior nodes for fast access. In figure 4 we see an example with three Join patterns and six fragments. The most interesting fragment is **B** as it is shared by two patterns **A&B** and **B&C&D**. This design allows us to trade the state-space explosion problem with a slightly longer matching time. However we further optimize the search time by only checking the part of the tree that is directly affected by the occurrence of the Join method fragment call. Using our example from the figure when a **C** is called only **B&C&D** is checked for completion. If **B** is called both **A&B** and **B&C&D** are checked for completions. This is achieved by connecting the leaves to multiple interior nodes so that when a fragment arrives it is quick to check if that new fragment completes any Join patterns. In the pattern matcher a list of all fragments are stored in the root of the node so that when a call arrives we can immediately access the correct location in the tree without the need to traverse the data-structure. In this way we optimize the pattern matching process to only search the part of the tree that contains patterns affected by the arrival of the new fragment. That is if a Join method call occurs it only checks patterns that contain that Join method fragment.

The Join calculus has a non-deterministic matching semantic on reduction of rules. However as related earlier, in the pattern matcher we have extended the semantics to support deterministic reduction. We did this via the **ordered** modifier. When the pattern matcher is in deterministic reduction mode it will match all possible patterns in the pool rather than the first randomly selected match. The pattern matcher will then choose which pattern to complete based upon the order in which they were defined in the Join class. The worst-case scenario for this pattern matcher is if a Join fragment occurs in every Join method. This will lead to every pattern being searched. We believe this is not likely to happen in the general case as most Join method fragments would have locality, that is most Join fragments only take part in a few Join patterns.

4.5 Precalculated Pattern Matcher

The second major pattern matcher we developed was designed to optimize the speed of execution for a limited number of fragments. This pattern matcher calculated every possible state that the pattern matcher could exist in and in the event of a change in the state space would immediately know the appropriate pattern to execute. The state of the pattern matcher is expressed as a series of bits used to represent an integer value. This integer value gives a position in the precalculated array that resolves to a completed pattern. The array is thus expressed as a linear array with a magnitude of 2^n where n is the number of fragments in the Join class. The state of the pattern matcher at any point in time can be expressed as a sequence of bits indicating the presence or absence of a particular fragment. For example, a Join class containing five fragments (**a** through **e**), there are 32 possible states *00000* through to *11111*. If there was an **a** and a **c** waiting the bitfield would be *10100*. The design of the precalculated pattern matcher is illustrated in figure 5. In the event that more than one fragment is waiting the bit field will still only be 1. Therefore 1 represents some (one or more) fragments waiting in the pattern matcher and 0 means no fragments waiting in the pattern matcher. Because the state can be converted into an index via trivial bit manipulation, retrieval of completed patterns is performed quickly. When initialization occurs (in the prototype when the first fragment is called), the pattern matcher calculates all possible states the pattern matcher could be in and calculates the resultant completed patterns from those states.

The major advantage of this approach is that the pattern matcher design has a constant delay when retrieving completed patterns. This is because after the precalculation of states is done no searching needs to be performed during runtime. The state of the pattern matcher is stored as a position in the precalculated array. Consequently the time fluctuations other matching algorithms suffer from is removed.

This pattern matcher has two disadvantages. Firstly, In the event of a large number of fragments (larger than 16 fragments) the precalculation period takes an increasingly long time to populate the array. However, as this can be done at compile time this penalty is not as great as first appearances would suggest. The second disadvantage is the memory footprint of the pattern matcher is relatively

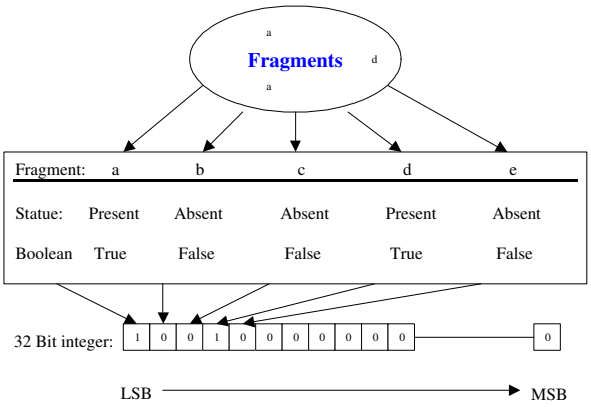

Fig. 5. Precalculated Pattern Matcher

large compared to previous implementations. The number of fragments that can be handled by the pattern matcher is limited by the memory that can be allocated to the precalculated array. The system has been tested using up to 16 Join fragments (requiring a 256k precalculated array).

Initial benchmarking of the pattern matcher has indicated that performance is adequate in the majority of cases. The greatest delay is the initial call that generates the precalculated table. However, as we stated earlier in any final implementation this would be done at compile time rather than runtime. Once the initial call has completed the speed of the method calls and the resultant pattern matching seems to be close to that of normal Java.

4.6 Symmetry

Most of the pattern matchers we have examined have had limitations in state space explosion or alternatively are expensive at runtime. Consequently it is interesting to look at what optimization we can make to the algorithms to improve the runtime speed. One such approach we have been looking at is that of symmetry. The idea of symmetry is to locate similar patterns within the pattern matcher and group them together. For example, if we had the following patterns **a()&b()**, **a()&c()**, **a()&d()** we would end up with 2^4 possible states. However, if we examine the patterns we see that all of them are fairly similar being of the form **a()&α()** where α is **b**, **c** or **d**. We could then store the pattern **a()&α()** in the pattern matcher. When a call to **b**,**c** or **d** occurs we store a call to α. Consequently the state space is limited (in this case) to 2^2 possible states hence reducing state space explosion. The disadvantage of this approach is we pay a penality of interpretation when we find a complete pattern as we need to figure out which fragment **a**, **b** or **c** has been called.

5 Conclusion and Future Work

In this paper we have provided a brief overview of the Join Java language. We showed how that with two small additions to the syntax of Java we have added a powerful concurrency synchronisation mechanism to the language. We then reflected on some of our experiences with implementing the Join Java compiler and runtime. We have shown how the language itself carries out a lot of the work synchronising interactions and handling communications without making the programmer worry about low-level operations (the use of **wait** and **notify** are virtually unnecessary as they are completely handled by the runtime system of the language). This should reduce the occurrence of errors in software increasing quality for little cost.

Join Java better represents concurrency as it allows other formalisms to be directly expressed in the language with a minimum change from the core Java language. The structure of Join Java allows message passing to be easily implemented between different processes without having to concern oneself with the low level details of the message passing implementation.

In every implementation of a Join type language the critical factor to the success of the language is the pattern matcher. Whilst designing the various pattern matchers we came to the conclusion that no single pattern matcher can ever efficiently solve all possible configurations of patterns and fragments. To this end we are spending considerable effort on this component of our compiler looking for novel solutions in order to increase the speed without compromising scalability, speed or memory size. This has proved difficult but we feel that our first few pattern matchers are a good start in this direction. We would like to thank Professor Martin Odersky and Matthias Zenger from Ecole Polytechnique Federale de Lausanne for their assistance with this research.

We are refining further pattern matcher algorithms so we can explore how to increase the speed of matching. We are also designing an artificial test-rig that will simulate the behaviour of a Join program at runtime so that performance can be measured and compared against different situations in a repeatable way. We are further developing a large set of Join Java programs so that we can do full regression testing on the compiler itself.

References

1. Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, Cambridge, MA, 1992.
2. Nick Benton, Luca Cardelli, and Cdric Fournet. Modern concurrency abstractions for csharp. In *in Proceedings of FOOL9*, 2002.
3. G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 1(96):217–248, 1992.
4. Peter A. Buhr. Are safe concurrency libraries possible? *Communications of the ACM*, 38(2):117–120, 1995.
5. C. Colby, L. Jagadeesan, R. Jagadeesan, K. Laufer, and C. Puchol. Design and implementation of triveni: A processalgebraic api for threads + events. In *International Conference on Computer Languages. 1998.* IEEE Computer Press, 1998.

6. Lobel Crnogorac, Anands Rao, and Kotagiri Romamohanarao. Classifying inheritance mechanisms in concurrent object-orinted programming. In Eric Jul, editor, *ECOOP'98 - European Conference on Object Oriented Programming*. Springer - Lecture Notes in Computer Science 1445,, 1998.
7. F. le Fessant and L. Maranget. Compiling join patterns. In U. Nestmann and B. C. Pierce, editors, *HLCL '98 in Electronic Notes in Theoretical Computer Science*, volume 16, Nice, France, 1998. Elsevier Science Publishers.
8. Luc Maranget F. L. Fessant and Sylvain Conchon. Join language manual, 1998.
9. C. Fournet, G. Gonthier, J. Lvy, L. Maranget, and D Rmy. A calculus of mobile agents. *Lecture Notes in Computer Science*, 1119, 1996.
10. Cedric Fournet and Georges Gonthier. The reflexive cham and the join-calculus. In *Proc. 23rd Annual ACM Symposium on Principles of Programming Languages*, volume January, pages 372–385. ACM Press, 1996.
11. Cedric Fournet, Cosimo Laneve, Luc Maranget, and Didier Remy. Inheritance in the join calculus. In S. Kapoor and S. Prasad, editors, *FST TCS 2000: Foundations of Software Technology and Theoretical Computer Science*, volume 1974, pages 397–408, New Delhi India, 2000. Springer-Verlag.
12. James Gosling and H McGilton. The java language environment, 1995.
13. Gerald Hilderink, Andre Bakkers, and Jan Broenink. A distributed real-time java system based on csp. In *Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, ISORC 2000*, pages 400–407, Newport Beach, California, 2000. IEEE.
14. C. A. R. Hoare. Communicating sequential processes. In R. M. McKeag and A. M. Macnaghten, editors, *On the construction of programs – an advanced course*, pages 229–254. Cambridge University Press, 1980.
15. C.A.R Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.
16. G Stewart Itzstein and Kearney David. Applications of join java. In *Proceedings of the Seventh Asia Pacific Computer Systems Architecture Conference ACSAC'2002*, pages 1–20, Melbourne, Australia, 2002. Australian Computer Society.
17. G Stewart Itzstein and David Kearney. Join java: An alternative concurrency semantic for java. Technical Report ACRC-01-001, University of South Australia, 1 January 2001 2001.
18. Luc Maranget, F. L. Fessant, and Sylvain Conchon. Jocaml manual, 1998.
19. S Matsuoka and A Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In P Agha, P Wegner, and A Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.
20. Martin Odersky. Functional nets. In *European Symposium on Programming*, volume 1782, pages 1–25, Berlin Germany, 2000. Springer Verlag.
21. Martin Odersky. Programming with functional nets. Technical 2000/331, Ecole Polytechnique Fdrale de Lausanne, March 2000 2000.
22. Guy L Steele. *Rabbit: A Compiler for Scheme*. Masters, MIT, 1978.
23. P. H. Welch. Java threads in the light of occam-csp. In P. H. Welch Bakkers and A. W. P., editors, *Architectures, Languages and Patterns for Parallel and Distributed Applications*, volume 52 April http://www.cs.ukc.ac.uk/pubs/1998/702 of *Concurrent Systems Engineering Series*, pages 259–284. WoTUG IOS Press, Amsterdam, 1998.
24. Matthias Zenger and Martin Odersky. Implementing extensible compilers. In *ECOOP 2001 Workshop on Multiparadigm Programming with Object-Oriented Languages*, Budapest, 2001.