Java as a Specification Language for Hardware-Software Systems

Rachid Helaihel and Kunle Olukotun
Computer Systems Laboratory
Stanford University, Stanford, CA 94305
{rashhel, kunle}@ogun.stanford.edu

Abstract

The specification language is a critical component of the hardware-software co-design process since it is used for functional validation and as a starting point for hardware-software partitioning and co-synthesis. This paper proposes the Java programming language as a specification language for hardware-software systems. Java has several characteristics that make it suitable for system specification. However, static control and dataflow analysis of Java programs is problematic because Java classes are dynamically linked. This paper provides a general solution to the problem of statically analyzing Java programs using a technique that pre-allocates most class instances and aggressively resolves memory aliasing using global analysis. The output of our analysis is a control dataflow graph for the input specification. Our results for sample designs show that the analysis can extract fine to coarse-grained concurrency for subsequent hardware-software partitioning and co-synthesis steps of the hardware-software co-design process to exploit.

1 Introduction

Hardware-software system solutions have increased in popularity in a variety of design domains [1] because these systems provide both high performance and flexibility. Mixed hardware-software implementations have a number of benefits. Hardware components provide higher performance than can be achieved by software for certain time-critical subsystems. Hardware also provides interfaces to sensors and actuators that interact with the physical environment. On the other hand, software allows the designer to specify the system at high levels of abstraction in a flexible environment where errors - even at late stages in the design - can be rapidly corrected [2]. Software therefore contributes to decreased time-to-market and decreased system cost.

Hardware-software system design can be broken down into the following main steps: system specification, partitioning, and co-synthesis. The first step in an automatic hardware-software co-design process is to establish a complete system specification. This specification is used to validate the desired behavior without considering implementation details. Functional validation of the system specification is critical to keep the system development time short because functional errors are easier to fix and less costly to handle earlier in the development process. Given a validated system specification, the hardware-software partitioner step divides the system into hardware, software subsystems, and necessary interfaces by analyzing the concurrency available in the specification. The partitioner maps concurrent blocks into communicating hardware and software components in order to satisfy performance and cost constraints of the design. The final co-synthesis step generates implementations of the different subsystems by generating machine code for the software subsytems and hardware configuration data for the hardware subsystems.

The system specification is a critical step in the co-design methodology because it drives the functional validation step and the hardware-software partitioning process. Thus, the choice of a specification language is important. Functional validation entails exploration of the design space using simulation; hence, the specification must allow efficient execution. This requires a compile and run-time environment that efficiently maps the specification onto general-purpose processor platforms. On the other hand, the partitioning process requires a precise input specification whose concurrency can be clearly identified. Generating a precise specification requires language constructs and abstractions that directly correspond to characteristics of hardware or software. Traditionally, designers have not been able to reconcile these two objectives in one specification language, but have instead been forced to maintain multiple specifications. Obviously, maintaining multiple specifications of the design is at best tedious due to the need to keep all specifications synchronized. It is also error-prone because different specification languages tend to have different programming models and semantics. This need for multiple specifications is due to shortcomings of

0-89791-993-9/97 $10.00 © 1997 IEEE

current specification languages used in hardware-software co-design.

Hardware-software specification languages currently used by system designers can be divided into software programming languages and hardware description languages. Software languages such as C or C++ generate high-performance executable specifications of system behavior for functional validation. Software languages are traditionally based on a sequential execution model derived from the execution semantics of general purpose processors. However, software languages generally do not have support for modeling concurrency or dealing with hardware issues such as timing or events. These deficiencies can be overcome by providing the designer with library packages that emulate the missing features [15]. A more serious problem is that software languages allow the use of indirect memory referencing which is very difficult to analyze statically. This makes it difficult for static analysis to extract implicit concurrency within the specification. Hardware description languages such as Verilog [5] and VHDL [6] are optimized for specifying hardware with support for a variety of hardware characteristics such as hierarchy, fine-grained concurrency, and elaborate timing constructs. Esterel is another specification language similar to Verilog with more constructs for handling exceptions [7]. SpecCharts builds on a graphical structural hierarchy while using VHDL to specify the implementations of the various structures in the hierarchy [4]. These languages do not have high-level programming constructs, and this limits their expressiveness and makes it difficult to specify software. Furthermore, these languages are based on execution models that require a great deal of run-time interpretation such as event-driven semantics. This results in low-performance execution compared to software languages.

This paper advocates the use of Java as a single specification language for hardware-software systems by identifying key language characteristics that enable both efficient functional validation and concurrency exploration by the hardware-software partitioner. Java is a general-purpose, concurrent, object-oriented, platform-independent programming language [10]. Java is implementation-independent because its run-time environment is an abstract machine called the Java virtual machine (JVM) with its own instruction set called bytecodes [11]. The virtual machine uses a stack-based architecture; therefore, Java bytecodes use an operand stack to store temporary results to be used by later bytecodes. Java programs are set in an object-oriented framework and consist of multiple classes, each of which is compiled into a binary representation called the classfile format. This representation lays out all the class information including the class's data fields and methods whose code segments are compiled into bytecodes. These fields and methods can be optionally declared as static. Static fields or methods of a class are shared by all instances of that class while non-static fields or methods are duplicated for each new instance. Data types in Java are either *primitive types* such as integers, floats, and characters or *references* (pointers) to class instances and arrays [10]. Since Java classes are predominantly linked at run-time, references to class instances cannot be resolved at compile-time. This presents a challenge to static analyzers in determining data flow through data field accesses and control flow through method calls.

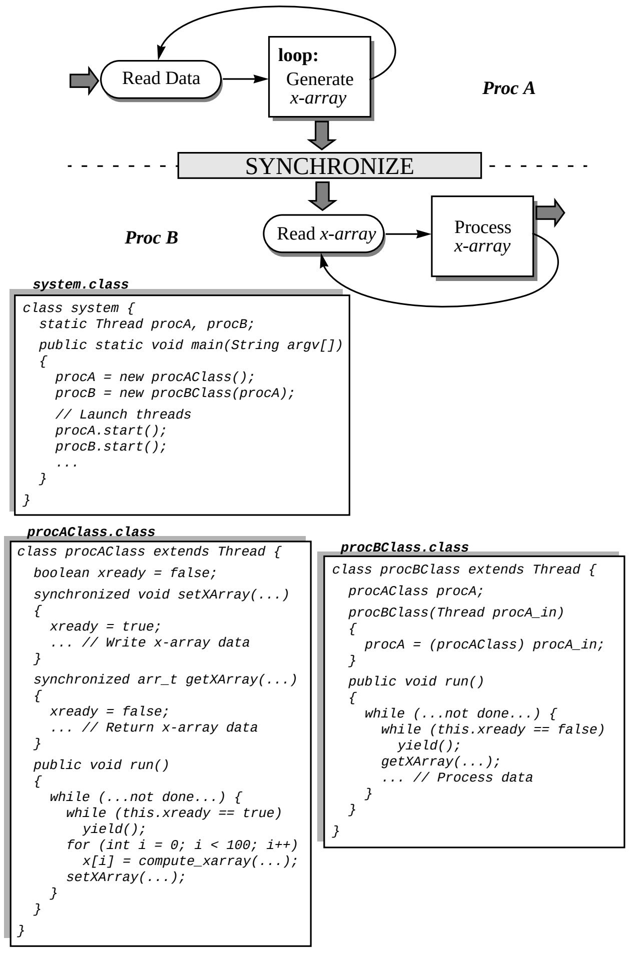
This paper also outlines a control/dataflow analysis technique that can be used as a framework for detecting concurrency in the design. Our analysis technique provides a general solution for the problem of dynamic class allocation by aggressively pre-allocating most class instances at compile-time and performing global reference analysis.

The rest of the paper is organized as follows. In Section 2 we explain why Java is well-suited for hardware-software system specification. In Section 3 we identify the problems that arise when analyzing Java programs and present a general solution for building control flow and dataflow dependence information. We apply our technique to three sample designs and analyze both explicit and implicit concurrency in these designs in Section 4. We conclude and briefly discuss future directions in Section 5.

2 Hardware-Software Specification with Java

It is desirable for the hardware-software co-design process to use a single specification language for design entry because specifications using different languages for software and hardware combine different execution models. This makes these specifications difficult to simulate and to analyze. Some researchers begin with a software programming language usually C++ and extend this language with constructs to support concurrency, timing, and events by providing library packages or by adding new language constructs. Examples of these approach are Scenic [15] and V++ [17]. We take a slightly different approach. Instead of requiring the designer to specify the hardware implementation details in the specification, in our approach the designer models the complete system in an algorithmic or behavioral fashion. Software languages are well-suited for this type of modeling. Once the specification is complete, an automatic compilation process is used to analyze the specification to identify the coarse-grained concurrency described by the designer and uncover the finer-grained concurrency implicit in the specification. The partitioning and synthesis steps of the hardware-software co-design process use the concurrency uncovered by this analysis to create an optimized hardware-software system. The specification language used with this approach must have the ability to specify explicit concurrency and make it easy to uncover the implicit concurrency.

Coarse-grained concurrency is intuitive for the designer to specify because hardware-software systems are often conceptualized as sets of concurrent behaviors [2]. Java is a multi-threaded language and can readily express this sort of concurrency. Such concurrent behaviors can be modeled by sub-classing the *Thread* class and overriding its *run* method to encode the thread behavior as shown in Figure 1.



system.class
```
class system {
  static Thread procA, procB;
  public static void main(String argv[])
  {
    procA = new procAClass();
    procB = new procBClass(procA);
    // Launch threads
    procA.start();
    procB.start();
    ...
  }
}
```

procAClass.class
```
class procAClass extends Thread {
  boolean xready = false;
  synchronized void setXArray(...)
  {
    xready = true;
    ... // Write x-array data
  }
  synchronized arr_t getXArray(...)
  {
    xready = false;
    ... // Return x-array data
  }
  public void run()
  {
    while (...not done...) {
      while (this.xready == true)
        yield();
      for (int i = 0; i < 100; i++)
        x[i] = compute_xarray(...);
      setXArray(...);
    }
  }
}
```

procBClass.class
```
class procBClass extends Thread {
  procAClass procA;
  procBClass(Thread procA_in)
  {
    procA = (procAClass) procA_in;
  }
  public void run()
  {
    while (...not done...) {
      while (this.xready == false)
        yield();
      getXArray(...);
      ... // Process data
    }
  }
}
```

Figure 1. Concurrency in Java

The *Thread* class provides methods such as *suspend* and *resume*, *yield*, and *sleep* that manipulate the thread. Synchronization, however, is supported at a lower level using monitors implemented in two bytecode operations that provide an entry and an exit to the monitor. The sample design shown in Figure 1 maintains synchronization when reading and writing the *x-array* in methods *getXArray* and *setXArray* which are tagged as *synchronized*.

Fine-grained concurrency is usually either non-intuitive or cumbersome for the designer to express in the specification. This implies that an automated co-design tool must be able to uncover fine-grained concurrency by analyzing the specification. The primary form of concurrency to look for is loop-level concurrency where multiple iterations of the same loop can be executed simultaneously. This form of concurrency is important to detect because algorithms generally spend most of their time within core loops. Identifying and exploiting parallel core loops can thus provide significant performance enhancements. Determining whether loop iterations are parallel requires analysis to statically determine if data dependencies exist across these loop iterations. In the *run* method of *procAClass* shown in Figure 1, if the *compute_xarray* call does not depend on values generated in previous iterations of the for-loop, then all the iterations of the loop may be executed simultaneously. The major hurdle that the data dependence analysis must overcome is dealing with memory references because these references introduce a level of indirection in reading and writing physical memory locations. Compile-time analysis has to be conservative in handling such references. This conservatism is necessary to guarantee correct system behavior across transformations introduced by the partitioning step based on the results of the analysis. However, this conservatism causes the analysis to generate false data dependences which are nonexistent at the system level. These dependences reduce the data parallelism that the analysis detects. In the simple design shown in Figure 1, without the ability to analyze dependences within the for-loop and across the associated method call, conservative analysis would determine that the loop iterations are interdependent and hence can only be performed sequentially reducing the degree of data parallelism in that section of the specification by 100-fold. The advantage that Java has over a language like C++ is that Java restricts the programmer's use of memory references. In Java, memory references are strongly typed. Also, references are strictly treated as object handles and not as memory addresses. Consequently, pointer arithmetic is disallowed. This restrictive use of references enables more aggressive analysis to reduce false data dependences.

A co-design specification language should provide high-performance execution to enable rapid functional validation. Java's execution environment uses a virtual machine. The JVM provides platform-independence; however, this independence requires Java code to be executed by an interpreter which reduces execution performance compared to an identical specification modeled in C++. Although this performance degradation is at least an order of magnitude for Sun's JDK 1.0 run-time environment, techniques such as just-in-time compilation are closing the performance gap to less than two-times that of C++[12][16]. This evolution in Java tools and technology has been and will be driven by Java's success in other domains, especially network-based applications. Moreover, the Java run-time environment makes it easy to instrument and gather profiling information which can be used to guide hardware-software partitioning.

3 Analyzing Java Programs

Control and dataflow analysis of the Java specification is required for partitioning and co-synthesis steps of the co-design process. This analysis examines the bytecodes of invoked methods to determine their relative ordering and data dependencies. These bytecodes have operand and result types that are either primitive types, or classes and arrays. While primitive types are always handled *by value*, class and array variables are handled *by reference*. These object (class instance) references are pointers; however, they are well-behaved compared to their C/C++ counterparts because these references are strongly typed and cannot be manipulated.

Object references point to class instances that are linked dynamically during run-time. So, prior to executing the Java program, we can only allocate the static fields and methods of the program's classes. This makes it difficult to statically analyze Java programs because if object references cannot be resolved, calls to methods of these dynamically linked objects cannot be resolved either. This makes it impossible to determine control flow. The only way to deal with this problem is to conservatively assign the method invocation to software so that the software run-time system can handle the dynamic resolution. However, this reduces the opportunities for extracting parallelism in hardware and thus leads to inferior hardware-software design.

In order to avoid the problem with dynamically linked objects, the specification could be restricted to use only static fields and methods or be forced to allocate all necessary objects linearly at the beginning of the program. However, this would significantly restrict the use of the language. Our solution is to attempt to *pre-allocate* objects during static analysis. It should be noted that this approach does not handle class instantiations within loops or recursive method invocations.

Pre-allocation only partially solves the problem with dynamically allocated class instances. A class reference can point to any instance of compatible class type; therefore, two references of compatible class types can *alias*. Conservative handling of reference aliasing reduces the apparent concurrency in the specification. More aggressive reference aliasing analysis requires global dataflow analysis to determine a class instance or set of instances that a reference may point to.

```
ProcessMethod (current_method) {
    Perform local analysis on current_method to build local control
        flow information and resolve local dependencies.
    Pre-allocate new instantiations if not inside loops or recursion.
    For each method invoked {
        ProcessMethod (invoked_method)
        Resolve reference global analysis impacted by invoked_method
    }
    Resolve global dependencies given complete reference analysis
}

ProcessMethod (main)
```

Figure 2. Analysis technique outline

An outline of our analysis technique is shown in Figure 2. The analysis starts with the static *main* method. For each method processed, local analysis is performed to determine local control and dataflow. Next, all methods invoked by the current method are recursively analyzed. Finally, reference point-to values are resolved in order to determine global data dependence information. Before elaborating on the techniques used to perform the local and global analyses, we describe the target representation of the CDFG.

The CDFG representation shown in Figure 3 involves two main structures. The first structure is a table of static and pre-allocated class instances. Aside from object accounting information, this table maintains a list of entries per object; each entry represents either a method or a non-

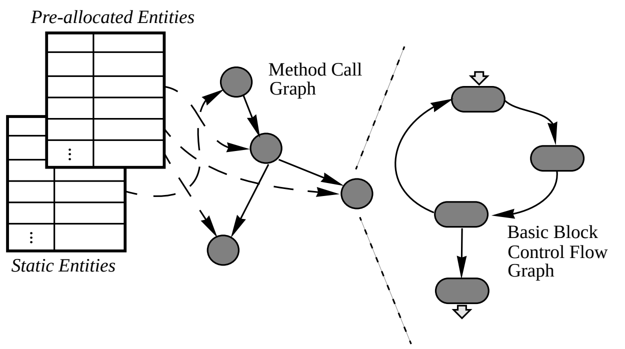
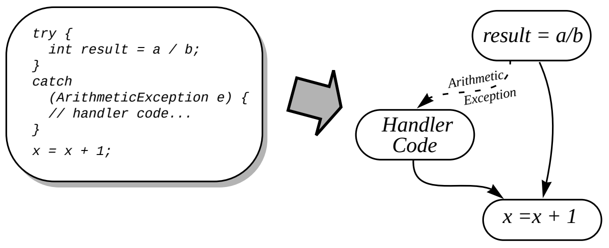


Figure 3. Target representation



```
try {
   int result = a / b;
}
catch
   (ArithmeticException e) {
   // handler code...
}
x = x + 1;
```

Figure 4. Exception edges in CDFG

primitive type data field. The data field entry is necessary for global analysis because data fields have a global scope during the life of their instances. Arrays are treated exactly as class instances. In fact, arrays are modeled as classes with no methods. The method entries point to portions of the second main structure in the representation. The second structure is the control dataflow information. Its nodes are bytecode basic blocks. The edges represent local control flow between basic blocks within a methods as well as global control flow across method invocations and returns.

The CDFG representation models multi-threading and exceptions using special control flow edges that annotate information about the thread operation performed or the exception trapped. Thread operations in Java are implemented in methods of the *Thread* class. The CDFG abstracts invocations of these methods by encoding the associated operation in the control flow edge corresponding to the method call. For example, Java threads are initiated by invoking *Thread* class *start* method. When the CDFG encounters an invocation of the start method, a new control flow edge is inserted between the invocation and the start of the thread's *run* method. This edge also indicates that a new thread is being forked. On the other hand, Exceptions in Java use *try-catch* blocks where the code which may cause an exception is placed inside the *try* clause followed by one or more subsequent *catch* clauses. Catch blocks trap on a specified *thrown* exception and execute the corresponding handler code. The CDFG inserts special control flow edges between the block that may cause the exception and the handler block. These edges are annotated with the type of exception the handler is trapping. An example of how exceptions are handled in shown in Figure 4.

3.1 Local Analysis

This step targets a particular method, identifying and sequencing its basic blocks to capture the local control flow. It also resolves local dependencies at two distinct levels. First, since Java bytecodes rely on an operand stack for the intermediate results, the extra level of dependency indirection through the stack needs to be factored out. This is achieved using *bytecode numbering*. Second, dependencies through local method variables are identified using reaching definition dataflow analysis.

Local Control Flow. Local control flow is represented by the method's basic blocks and the corresponding sequencing. Basic blocks are sequences of bytecodes such that only the first bytecode can be directly reached from outside the block and if the first bytecode is executed, then all the bytecodes are sequentially executed. The control flow edges simply represent the predecessor-successor ordering of all the basic blocks.

Local Dataflow Analysis. Dependencies exist between bytecodes through an extra level of indirection - the operand stack. We resolve this indirection by using "bytecode numbering." Bytecode numbering simply denotes the replacement of the stack semantics of each bytecode analyzed with physical operands that point to the bytecode that generated the required result. This is simply achieved by traversing the method's bytecodes in program order. Instead of executing the bytecode, its stack behavior is simulated using a compile-time operand stack, *OpStack*. If the bytecode reads data off the stack, entries are popped off OpStack, and new operands are created with the values retrieved from the stack. If the bytecode writes a result to the stack, a pointer to it is pushed onto OpStack. This process has to account for data that requires more than one stack entry such as double precision floating point and long integer results. Also, stack-manipulating bytecodes such as *dup* (duplicate top entry) or *swap* (swap top two entries) are interpreted by manipulating OpStack accordingly. Then, these bytecodes are discarded since they are no longer needed for the purposes of code functionality. An outline and an example of bytecode numbering are shown in Figure 5.

Data dependencies across local variables are resolved by computing the reaching definitions for the particular method. A definition of a variable is a bytecode that may assign a value to that variable. A definition *d* reaches some point *p* if there exists a path from the position of *d* to *p* such that no other definition that overwrites *d* is encountered. Once all the reaching definitions are computed, it would be clear that there exists a data dependency between bytecode *m* and bytecode *n* if m defines a local variable used by *n* and *m*'s definition reaches the point immediately following *n*.

Computing the reaching definitions uses the iterative dataflow Worklist algorithm [8]. This algorithm iterates over all the basic blocks. A particular basic block propa-

```
Initialize symbolic operand stack, OpStack, to empty.
Traverse basic blocks in reverse postorder.
    If current bytecode reads data from the stack,
        pop OpStack into the appropriate bytecode
            operand slot.
    If current bytecode writes data to the stack,
        push the bytecode's PC unto OpStack.
```

```
0    iload_1       push local variable onto operand stack
1    ldc_w #4      push constant onto operand stack
4    if_icmplt 9   pop two entries, if first < second -> jump to PC= 9
7    iconst_1      push constant = 1
8    ireturn       pop entry and return from method with entry as return value
9    iconst_m1     push constant = -1
10   ireturn       pop entry and return from method with entry as return value
```

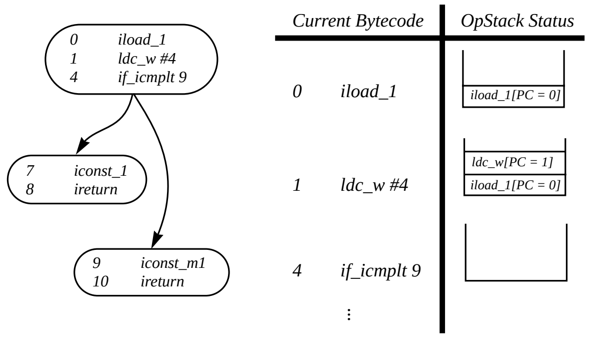


Figure 5. Bytecode numbering example

gates definitions it does not overwrite. At a join point of multiple control branches, the set of reaching definitions is the union of the individual sets. The algorithm iterates over the set of successors of all basic blocks whose output set of reaching definitions changes and converges when no more changes in these sets of reaching definitions materialize.

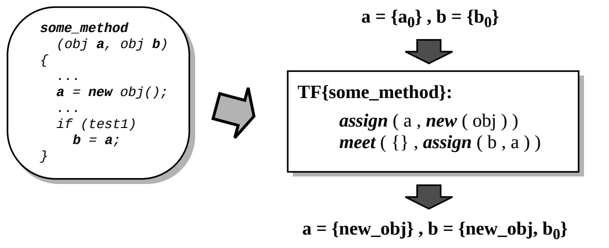
3.2 Global Analysis

To handle data dependencies between references, global analysis generates for each reference the set of object instances to which it may point out of the set of pre-allocated instances. Once this points-to relation is determined, simple dataflow analysis techniques such as global reaching definition can compute dataflow dependencies between these references.

A straightforward solution is to examine the entire control flow graph while treating method invocations as regular control flow edges. Then, iterative dataflow analysis can generate the points-to information for every reference. However, this approach suffers from the problem of unrealizable paths which cause global aliasing information to propagate from one invocation site to a non-corresponding return site [9].

A more context-sensitive solution motivated by [9] is to generate a transfer function for each method to summarize the impact of invoking that method on globally accessible data and references. The variables that this transfer function maps are the formal method parameters that are references. In addition, this set of variables is extended to include global references used inside the method through (1) creating new instances, (2) invoking other methods that return object references, or (3) accessing class instance fields that are references. Input to this transfer function is the initial points-to values of the extended parameters set. Output generated by this transfer function is the final points-to values of the extended parameters due to the method invocation.

This transfer function is a summary of the accesses (reads and writes) of the method's extended parameters generated using interval analysis [8]. These accesses are ordered according to the method's local control flow information. Accesses can be one of the following five primitive operations: *read*, *assign*, *new*, *meet* and *invoke*. The *read* primitive requires one operand which is a reference; the result is the set of potential class instances to which the reference points. The *assign* primitive is used to summarize an assignment whose left-hand side is an extended parameter. It requires two operands the first of which is the target reference. The second is a set of potential point-to instances. The *new* primitive indicates the creation of a new class instance. This primitive returns a set composed of a single instance, if pre-allocation is possible (not within loop or recursion). Otherwise, it conservatively points-to the set of compatible class instances. The *meet* primitive is necessary to handle joining branches in the control flow. At a meet point, the alias set of some reference assigned in one or more of the meeting branches is the union of the alias sets for that reference from each of the meeting control flow edges. Finally, the *invoke* primitive is used to resolve change in reference alias sets due to invoking some method. Effectively, this primitive causes the transfer function of the invoked method to be executed.



a = {a0} , b = {b0}
TF{some_method}:
assign (a , new (obj))
meet ({} , assign (b , a))
a = {new_obj} , b = {new_obj, b0}

```
some_method
  (obj a, obj b)
{
  ...
  a = new obj();
  ...
  if (test1)
    b = a;
}
```

Figure 6. Transfer functions for global reference analysis

4 Experimental Results

Hardware-software systems are multi-process systems, so partitioning and co-synthesis tools which map behavioral specifications to these systems need to make hardware-software trade-offs [13]. To make these trade-offs with the objective of maximizing the cost-performance of the mixed implementation, it is necessary to be able to identify the concurrency in the input specification. We have implemented our Java front-end analysis step as a stand-alone compilation pass that reads the design's class files and generates a corresponding CDFG representation. We tested our technique using the designs listed in Table 1.

	Lines of Java	Classes	Instances	Basic Blocks
raytracer	698	6	37	358
robotarm	1325	10	53	583
decoder	3177	23	136	2342

Table 1: Design characteristics

The first design, *raytracer*, is a simple graphical application. It renders two spheres on top of a plane with shadows and reflections due to a single, specular light source. The second application, *robotarm*, is a robot arm controller. The third design, *decoder*, is a digital signal processing application featuring a video decoder for H.263-encoded bitstreams [14].

The resulting control-dataflow graphs were analyzed to identify concurrency in the specification. The analysis examined concurrency at three levels: thread-level, loop-level, and bytecode-level. Thread-level concurrency is exhibited as communicating, concurrent processes which can span the control flow of several methods. Loop-level concurrency is exhibited by core loops usually confined to a single method. Bytecode-level concurrency is exhibited by bytecode operations that can proceed provided their data dependencies are satisfied irrespective of a control flow ordering. This form of concurrency exists within basic blocks.

Thread-level concurrency is explicitly expressed by the designer through Java threads. Since threads are uniquely identified in the CDFG, no work is required to uncover this form of parallelism. Loop-level concurrency requires analysis of control and dataflow information associated with inner loops to identify data dependencies spanning different loop iterations and determine if these are true dependencies, that is, dependencies between a write in some iteration of the loop and a read in a subsequent iteration. So, loops with independent iterations can execute these iterations concurrently as mini-threads. The coarse-grained concurrency expressed at the thread or loop level can be exploited by allocating these threads to different subsystems in our target architecture.

On the other hand, bytecode-level concurrency in the CDFG does not span multiple basic blocks; it exists at the bytecode level within each basic block. Its degree depends on the basic block's "inter-bytecode" data dependencies. This fine-grained concurrency impacts the performance improvement of a hardware implementation of the basic block. Hardware is inherently parallel; therefore, parallelism in the design is implemented without any cost overhead given enough structural resources to support the parallelism. The only limitation on the degree of parallelism is synchronization due to data dependencies. Hence the execution time of some block in hardware decreases with increased data parallelism.

Table 2 presents the results of analyzing the three different forms of parallelism in our sample designs . The first column indicates the number of designer-specified threads. The second column shows the number of parallelizable loops while the third column indicates the average number of bytecodes per loop. The fourth column shows the average number of bytecodes per basic block while the fifth column assesses the average data parallelism in these basic blocks. This bytecode-level concurrency is measured as the average number of bytecodes that can execute simultaneously during a cycle of the JVM. These results show that it is possible to extract parallelism at various levels of granularity for Java programs.

	Thread-level Concurrency	Loop-level Concurrency		Bytecode-level Concurrency	
	Number of threads	Number of loops	Avg. bytecodes per loop	Avg. bytecodes/ basic block	Avg. bytecode parallelism
raytracer	2	10	20	6.6	2.0
robotarm	3	9	31	6.9	2.1
decoder	3	28	27	7.1	2.5

Table 2: Parallelism assessment results

5 Conclusions and Future Work

The specification language is the starting point of the hardware-software co-design process. We have described key requirements of such a language. A specification language should be expressive so that design concepts can be easily modeled but should provide a representation that is relatively easy to analyze and optimize for performance. The language should also provide high-performance execution. We have shown that the Java programming language satisfies these requirements.

To be able to partition and eventually co-synthesize input Java specifications, we must be able to analyze the specification. However, a major problem facing this analysis step in Java are dynamic links to class instances. To make static analysis possible, we proposed a technique that relies on aggressive reference analysis to resolve ambiguity in global control and dataflow. This technique generates a control dataflow graph representation for the specification. Our results show that using this technique it is possible to extract concurrency which can be exploited from the Java specification.

In the future, our analysis technique will serve as a front-end to a co-design tool which maps the Java system specification to a target architecture composed of one or more microprocessors tightly coupled to programmable hardware resources.

Acknowledgments

This work was sponsored by ARPA under grant no. MIP DABT 63-95-C-0049.

References

[1] J. Henkel, F. Vahid, and L. Ramachandran. *Hardware/Software Codesign of Embedded Systems Tutorial*. ICCAD 1995.

[2] M. Horowitz and K. Keutzer. “Hardware-Software Co-Design,” in the *Proceedings of the Synthesis and Simulation Meeting and International Interchange* (SASIMI), 1993.

[3] D. Gajski and F. Vahid. “Specification and Design of Embedded Hardware-Software Systems,” in *IEEE Design & Test of Computers*, pp. 53-67, Spring 1995.

[4] D. Gajski, F. Vahid, S. Narayan, and J, Gong. *Specification and Design of Embedded Systems*. Prentice Hall, 1994.

[5] D. Thomas and P. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 1991.

[6] IEEE Inc., NY. *IEEE Standard VHDL Language Reference Manual*, 1988.

[7] G. Berry and G. Gonthier. “The Esterel Synchronous Programming Language: Design, Semantics, Implementation,” *Science of Computer Programming* vol. 19, n˚2, pp 87-152, 1992.

[8] A. Aho, R. Sethi, and J. Ullman. *Compilers Principles, Techniques and Tools*. Addison-Wesley, 1986.

[9] R. Wilson and M. Lam. “Efficient Context-Sensitive Pointer Analysis for C Programs,” in the *Proceedings of the Conference on Programming Language Design and Implementation*, June 1995.

[10] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[11] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[12] B. Case. “Java Performance Advancing Rapidly,” in *Microprocessor Report*, pp. 17-19, vol. 10, issue 7, May 27, 1996.

[13] J. Adams and D. Thomas. “Multiple-Process Behavioral Synthesis for Mixed Hardware-Software Systems,” in the *Proceedings of the International Symposium on System Synthesis*, 1995.

[14] Telenor, Norway. *Enhanced H.263*. at http://www.fou.telenor.no/brukere/DVC/mpeg4/H.263+.html.

[15] S. Liao, S. Tjiang, and R. Gupta. “An Efficient Implementation of Reactivity for Modeling Hardware in Scenic Design Environment,” in the *Proceedings of the 34th Design Automation Conference*, June 1997.

[16] T. Cramer, et al. “Compiling Java Just in Time,” in *IEEE Micro*, pp. 36-43, Vol. 17, No. 2, May-June 1997.

[17] R. McGeer. “The V++ Systems Design Language,” a talk sponsored by the Stanford CAD group, May 1997.