Java Compilation for Multi-threaded Architectures

D.K. Arvind, J. Hossell, A. Koppe & R. Rangaswami
Institute for Computing Systems Architecture
Division of Informatics
The University of Edinburgh
King's Buildings, Mayfield Road
Edinburgh EH9 3JZ
Scotland - U.K.

Abstract

This paper outlines the design of a compilation framework for applications in embedded systems programmed in Java and targeted at multi-threaded architectures. These architectures have multiple Thread Processing Units (TPUs) to support loop-level parallelism, where each TPU employs instruction-level parallelism, and data and control speculation techniques to improve performance. The paper describes the compilation framework, with particular emphasis on the design of the Java Intermediate Representation (JIR) based on Static Single-Assignment (SSA). The framework cleanly separates target-independent compilation issues from target-specific ones, which enables the comparative study of multi-threaded models of computation.

1 Introduction

A Multi-threaded Architecture (MTA) comprises of several Thread Processing Units (TPUs) which mainly exploit loop-level parallelism by executing multiple flows of control [11, 7, 12, 9, 10]. Each TPU, in turn, supports Instruction Level Parallelism (ILP), control and data speculation, and efficient inter-thread communication and synchronisation. Research projects have designed compilers for multi-threaded targets [4, 8], which have mainly concentrated on applications programmed in C and Fortran.

This paper focuses on the compilation of embedded systems programmed in Java and aimed at multi-threaded targets. We propose exposing concurrency in the applications in the following three stages:

Source-level Application threads (i.e. Java threads) could potentially be mapped on to multi-processors, in which each processor is itself multi-threaded.

Intermediate-level At the next level, we identify concurrency in loops, conditionals and method-calls. Both control and data speculation are supported. The Java program is compiled to an intermediate representation form for analysis. This representation is independent of the underlying targets.

Low-level The final stage exploits instruction-level concurrency in the assembly code specific to the architecture.

This paper focuses on the second stage. Cost analysis (see Section 6) is used to determine concurrency within a source-level thread based on annotations in the Java Intermediate Representation (JIR). It can be used to guide the choice of certain features of the architecture, such as the optimum number of TPUs for the application. It is only in third stage that architecture-specific details are introduced.

The wider goal of the research project is to incorporate the compiler within a design framework for mobile embedded applications programmed in Java and targeted at multi-threaded asynchronous processors [1].

The rest of the paper is organised as follows: Section 2 gives an overview of the compilation framework; Section 3 describes the details of the Java Intermediate Representation (JIR), followed by a simple example in Section 4 for generating the JIR from bytecode; Sections 5 and 6 discuss the thread annotations in the JIR and performance prediction, respectively.

2 Overview of the Compiler

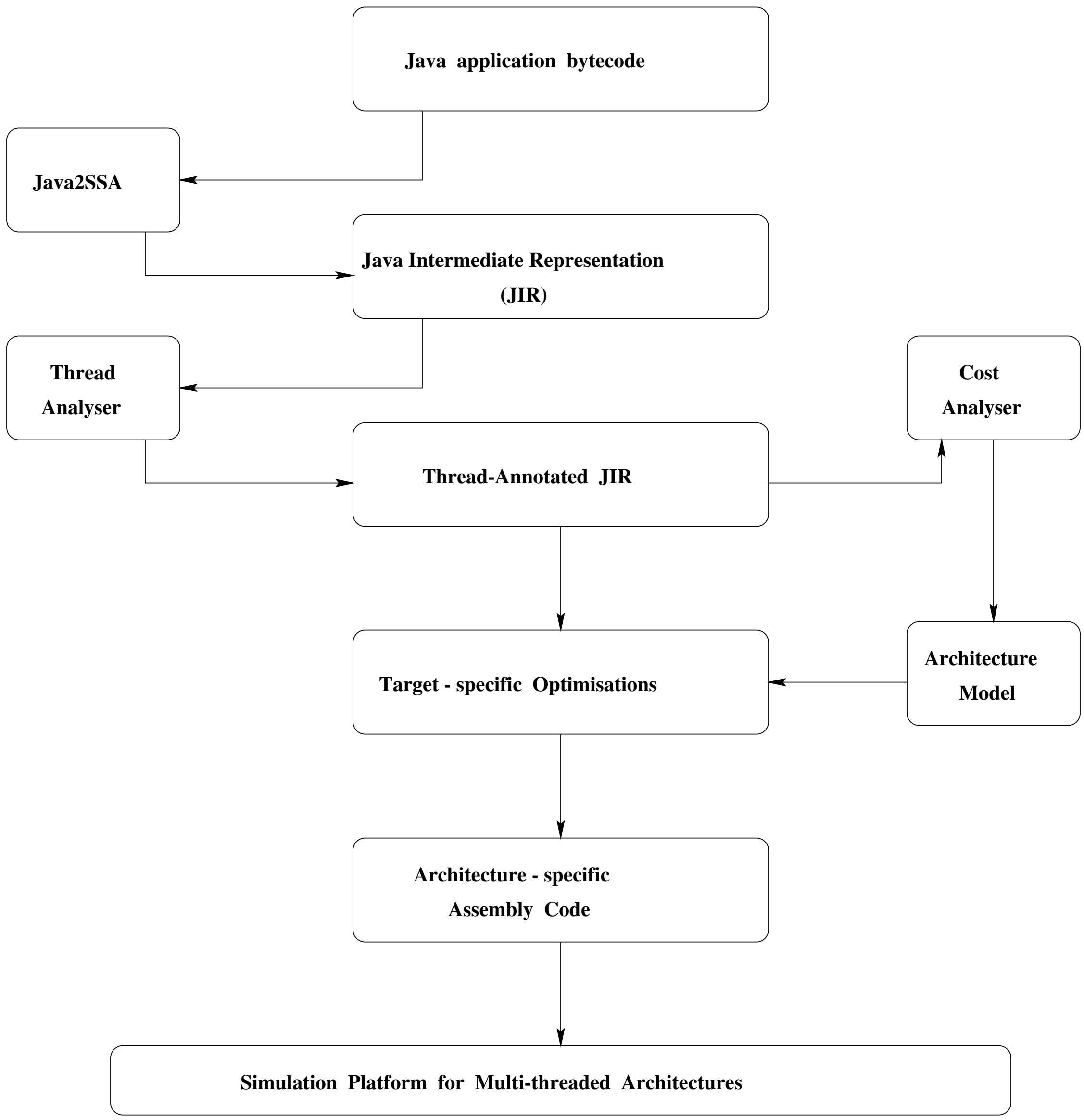

Figure 1: The Compilation Framework

Figure 1 illustrates the structure of the compilation framework. The Java application bytecode is first converted into an intermediate representation (JIR) which is based on the Static Single-Assignment (SSA) form [6, 3]. The JIR is suitable for both code optimisations [2], and multi-threading analysis, as it explicitly represents the dependences in the code. The *Thread Analyser* next examines the JIR for sources of potential concurrent execution, which at the moment is confined to program structures such as loops, conditionals and method calls [5]. The resulting SSA-version of the application code contains *annotations* at points where new threads are likely to be sparked. A cost and trace analysis is performed on the annotated code (see Section 6). This analysis could be refined to reflect the cost for a specific target, such as a Superthreaded architecture. The final stage generates architecture-specific assembly code which is executed on an architectural simulation of the multi-threaded target.

3 Java Intermediate Representation (JIR)

The Java Intermediate Representation (JIR) is a fully typed C++ class library which represents Java programs in SSA. The JIR preserves Java constructs at a high level of abstraction to enable language-level code optimisations.

The main attribute of a program represented in SSA is that every variable has a *unique* compile-time definition (this is, however, not necessarily true at run-time, as variables defined within a loop can be executed several times). The SSA form simplifies optimisations as the definitions and uses of variables are explicitly identified.

The *Java2SSA* module (in Figure 1) translates Java class files into a JIR data structure. The use of class files as the input was decided for the following three reasons:

- It obviates the need for a fully-fledged compiler front-end comprising a lexer, parser and a semantic analyser. The Java class file is itself an intermediate format at about the same level of abstraction as the JIR. However, the class file format is badly suited for transformations thanks to its stack orientation and its handling of constants, such as the *class* or *field* references which are difficult to manipulate.
- Pre-compiled software (i.e. bytecode), for which no source code is available, may be included.
- It allows the use of programs written in languages other than Java, such as Ada, to be compiled into the Java class file format.

3.1 The JIR Library

The JIR is a hierarchical C++ class library (see Figure 2), which consists of a number of single classes as well as class hierarchies representing elements of the Java Virtual Machine (JVM). At the root of a JIR data structure is the class *Program*. A Java program consists of classes and interfaces, together with an entry point for its execution, i.e. the method *main*. Each Java class and interface is represented by a C++ class called *Class*, which in turn contains fields and methods.

Methods can be implemented in Java bytecode as a collection of basic *Block*s connected by *Link*s, or they can be *abstract* or *native*. The *JavaMethod* represents the former and is more interesting for our purposes.

In turn, the *block* consists of definitions of a number of *phi functions*, followed by *basic operations*, and finally, an *exit* from the block.

The Phi functions are central to the SSA form. They *join* the variables from the different entry paths to a basic block into a single one. At run-time the result of a phi function is the argument that was defined most recently.

The basic operations define new variables in terms of others, create objects, access object fields and arrays, and call methods.

An *Exit* from a basic block can be any one of the following: a simple jump statement to another basic block, a conditional branch, a switched branch, an exception throw, or a return from the entire method. There is only one point of exit, which is at the end of the basic block.

Operations use variables as their arguments, and the *Var* class maintains a list of a variable's use sites.

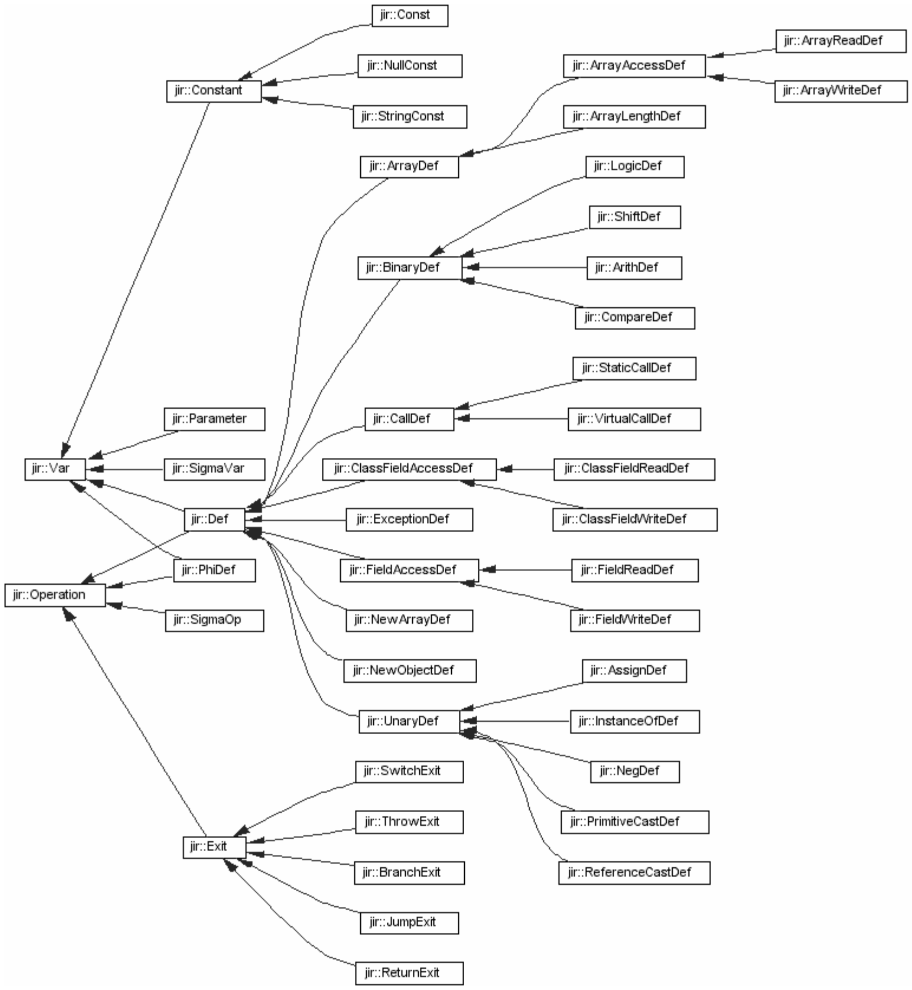

Figure 2: JIR class hierarchy

Definitions (class *Def*) in the JIR have a dual role: they are both *variables* (for use by other operations), as well as *operations* (as they use other variables).

The JIR is fully typed: every variable, constant, object field, method argument and return data has a type associated with it. The JVM type system is represented by the *Type* class hierarchy, and the *Class* class is itself part of it.

3.2 The Java2SSA translator

The `Java2SSA` module translates Java class files into the JIR: this is performed statically and does not support the translation of dynamic loading of additional class files at run-time. The first step is to parse the main class file, which, in turn, triggers the analysis of all the classes referenced recursively by the initial class. `Java2SSA` resolves constant pool entries and builds a JIR data structure representing the class hierarchy of the input program with all its fields and method signatures. The bytecode for each method is then translated in two passes.

1. The analysis of the control flow identifies the *basic blocks* by determining the points where *branches* and *joins* occur. The JVM subroutines are considerably more complicated and are dealt with by in-lining them, i.e. the jump-to-subroutine (*jsr*) and return (*ret*) instructions are treated as jumps. This does not impose too much of an overhead as the JVM specification restricts the use of these instructions: for instance, by disallowing recursion. The current version implements the exception, *throws*, but not, *catches*.

2. The stack-oriented execution model of the JVM is applied statically to the bytecode. The translation starts at a method's entry point and works individually through each basic block along the edges of the control flow graph. Each instruction is parsed only once, with the exception of those in subroutines, which due to in-lining may be parsed more than once. Experience with larger examples have shown that this does not cause problems in practice, although an exponential blow-up is theoretically possible.

   A dynamic representation of the stack reflects the current positions of variables and constants. It is updated accordingly as each JVM instruction is translated into the corresponding JIR operation. This task is relatively straightforward for linear code; however, when different control-flow paths *join* at the entry to a basic block, then the corresponding stack states have to be merged as part of the stack update operation: the stack states are compared and the differences between them are resolved by using the phi functions.

4 An example of generating the JIR

The translation of a Java program to its annotated JIR-form is elucidated with an example of a Vector-multiplication program (as shown in Figure 3).

The classfile was generated by the `javac` compiler from SUN Microsystems - an excerpt of a readable version of the `VecMult` class is shown in Figure 4, which was produced by `javap`, a Java disassembler utility.

The flow diagram on the right-hand side of Figure 4 illustrates the corresponding SSA form and its control flow graph (as generated by `Java2SSA`). Where possible, a value identifier refers to the name of the corresponding variable in the Java program. Although this relationship cannot be automatically reconstructed within the compiler, it has been used here to clarify the correspondence between the source program and its SSA version. Values labelled, t_n, refer to temporary values and as such do not have equivalents in the source code. The *rowVector* and *colVector* values are implicitly defined to be their arguments at the beginning of the method.

5 Thread Annotations in the JIR

The JIR version of the program is now amenable to threading analysis. At this stage we would ideally wish to incorporate *generic* multi-threading instructions, such as the *fork* and the *com-*

```
public class IntVec {

    public static int VecMult(int [] rowVector, int [] colVector)
    {
        int result = 0;

        for(int i = 0; i < colVector.length; i++)
            result += rowVector[i] * colVector[i];

        return result;
    }

    public static void main(String argv[])
    {

       int [] rowVec = new int[]{ 1, 2, 3, 4, 5};
       int [] colVec = new int[]{ 1, 1, 1, 1, 1};

       System.out.println("Result = " + VecMult(rowVec, colVec));
    }
}
```

Figure 3: A Vector-multiplication Program in Java

municate ones, which can then be implemented on any multi-threaded target. However, the multi-threading model of the target influences the different assumptions that have to be made regarding the structuring of the code as well as the level of architectural support available for threading. It is difficult in practice to use explicit multi-threading instructions in the JIR without inherently biasing it towards a particular model. In our experience much of the work towards inserting the apparently *generic* multi-threading instructions has to be undone, even for models favoured by the instructions.

For instance, the Superthreaded [11] and the Single Program Speculative Multi-threading (SPSM) [7] models treat different forms of concurrency. The former extracts parallelism from loop structures, while the latter deals with speculative parallelism in acyclic control flow (conditionals). Our approach exploits concurrency at different levels in the style of the Multiscalar project [9]:

- Method calls
- A single basic block
- A sub-graph of basic blocks, which contains one or more single basic blocks, which may contain a method call.

The JIR is *annotated* to split the control-flow graph generated by `Java2SSA` into a series of *tasks*, where a task is defined to contain one of the three previously-described items. This enables the complete graph to be broken into a series of schedulable units.

Given that a task may consist of a single method call, it is necessary to have tasks that do not start/finish at the beginning/end of a basic block. This leads to a succinct description of task execution:

- A task is started by calling its first instruction which is identified by an annotation.
- If the next instruction for execution is also the first instruction of (another) task, then the current instruction causes the (previous) task to terminate.

```
Method int VecMult(int[], int[])
   0 iconst_0
   1 istore_2
   2 iconst_0
   3 istore_3
   4 goto 20
   7 iload_2
   8 aload_0
   9 iload_3
  10 iaload
  11 aload_1
  12 iload_3
  13 iaload
  14 imul
  15 iadd
  16 istore_2
  17 iinc 3 1
  20 iload_3
  21 aload_1
  22 arraylength
  23 if_icmplt 7
  26 iload_2
  27 ireturn
```

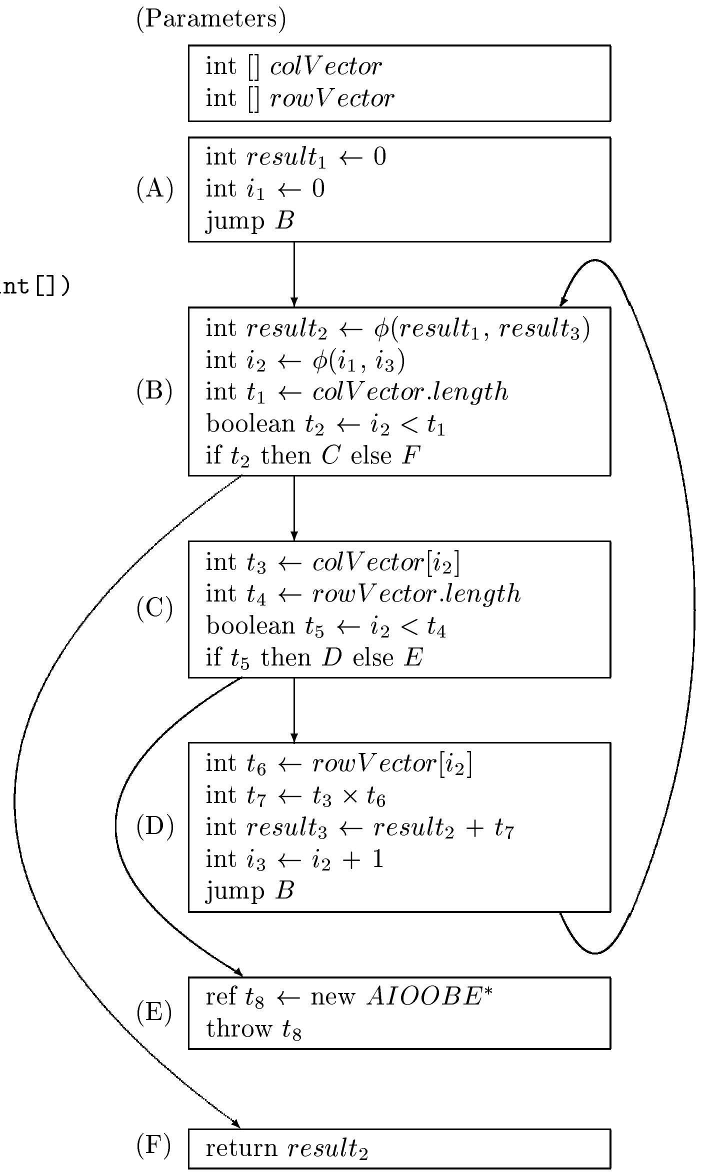

Figure 4: Translation of bytecode to SSA. (*java/lang/ArrayIndexOutOfBoundsException)

Such an approach offers a number of advantages:

- The entire program can now be described as a group of dependent but potentially parallel tasks. This is unlike either the Superthreaded or SPSM proposals which split the program into *multi-threaded* and *sequential* regions. In this approach the tasks can be structured to also exploit concurrency as envisaged by both the Superthreading and the SPSM models.
- The control flow graph accurately captures the control dependencies between the tasks, which enables both static and dynamic task scheduling.
- The use of SSA in the JIR expresses the data dependencies that exist between tasks which identifies the required inter-task communication.
- The approach - very importantly - provides a *common framework* in which an application can be evaluated on different multi-threading models. This is due to the fact that the task definition contains no references to communication or program structure. We can compare

the performance of the application for different multi-threaded models. For instance, the Multiscalar approach of exploiting loop-based concurrency, SESE (Single-Entry, Single-Exit) regions and method calls may incur a performance hit where this generality may be unnecessary; in which case, an alternative approach based on the Superthreaded or SPSM models may be better suited for the application.

- The *annotations* serve the dual purpose of facilitating model-independent transformations, and guiding the inclusion of model-specific threading facilities.

If we reconsider the Vector-multiply example, then splitting it into tasks to exploit the parallelism in the loop body yields the structure in Figure 5.

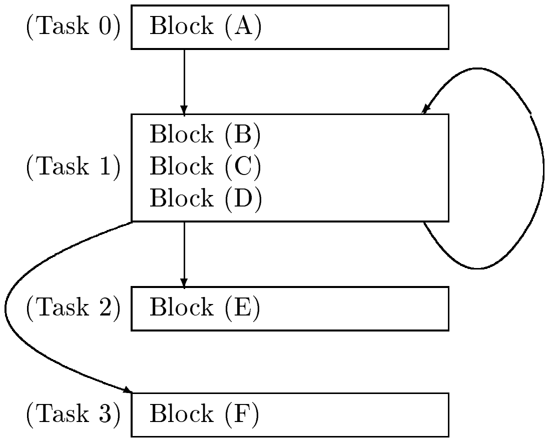

Figure 5: The task structure for the Vector-multiply program.

Task 1 may be scheduled a number of times depending on the value of the induction variable, which can all execute as concurrent threads.

6 Performance Prediction of Annotated JIR

The Cost Analyser predicts performance early in the compilation flow. The annotations referred to in Section 5 identify those parts of the program which offer scope for concurrency. Costs for computation, and thread management (forking and killing threads, and inter-thread communication) are defined which are then used to derive the overall cost for the program.

For a more detailed analysis, the task annotations, together with a sequential execution trace of the program, are used to derive a multi-threaded execution trace to address the following issues:

- Highlight dependencies that may degrade performance.

  A result calculated in thread n for use in thread $n + 1$ may delay the execution of the latter thread. Such delays can be targets for performance-boosting transformations.

- Realise the degree of concurrent threads for a given trace.

  The annotations will identify a maximum number of threads for the trace. However, the number of live threads at any instant will vary. These values can be identified in the multi-threaded execution trace to measure the effectiveness of transformations.

- Measure the utilisation of TPUs for different numbers of threads

  The efficiency of TPU usage can be gauged by measuring the number of instructions executing in parallel in the different threads.

7 Conclusions

This paper has described a compilation framework for Java applications which target multi-threaded architectures. A Java Intermediate Representation (JIR) based on Static Single-Assignment (SSA) facilitates concurrency analysis. A preliminary cost analysis together with a multi-threaded execution trace aids the choice of performance-boosting optimisations. The framework cleanly separates target-independent and target-specific issues in multi-threaded compilation, which enables the comparative study of multi-threaded models of computation.

Acknowledgement

The authors wish to acknowledge the support of SHEFC RDG grant No. 85 entitled *Design Clusters for System Level Integration*.

References

[1] D.K. Arvind and R. Rangaswami. Asynchronous multi-threaded processor cores for system level integration. In *Proceedings of IP'99*, November 1999.

[2] David F. Bacon, Susan L Graham, and Oliver J Sharp. Compiler transformations for high-performance computing. Technical report, Computer Science Division, University of California, Berkley, California, 1992.

[3] Marc M. Brandis and Hanspeter Mössenböck. Single-pass generation of static single-assignment form for structured languages. *ACM Transactions on Programming Languages and Systems*, Volume 16:pages 1684–1698, 1994.

[4] B.Zheng et al. Designing the agassiz compiler for concurrent multithreaded architectures. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing.*, August 1999. Agassiz compiler home page: http://www.cs.umn.edu/Research/Agassiz.

[5] Mike Chen and Kunle Olukotun. Exploiting method-level parallelism in single-threaded java programs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, October 1998. Hydra Home Page: http://ogun.stanford.edu/hydra.html.

[6] R. Cryton, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the sixteenth annual ACM symposium on Principles of programming languages*, pages pages 25–35, January 1989.

[7] Pradeep K. Dubey, Kevin O'Brien, Kathryn O'Brien, and Charles Barton. Single-program speculative multi-threading SPSM architecture: compiler-assisted fine-grained multi-threading. In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques*, volume PACT '95, pages 109–121, June 1995.

[8] Hideki Saito, Nicholas Stavrakos, Steven Carroll, Constantine Polychronopoulos, and Alex Nicolau. The design of the PROMIS compiler. In *Proceedings of the International Conference on Compiler Construction.*, 1999. PROMIS compiler home page: http://www.csrd.uiuc.edu/promis.

[9] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar processors. In *22th International Symposium on Computer Architecture*, 1995. Multiscalar Project Home Page: http://www.cs.wisc.edu/ mscalar/.

[10] The MAJC architecture homepage. The web-site contains presentation slides and white papers http://www.sun.com/microelectronics/MAJC/.

[11] Jenn-Yuan Tsai, Zhenzhen Jiang, and Pen-Chung Yew. Compiler techniques for the superthreaded architectures. In *International Journal of Parallel Programming - Special Issue on Languages and Compilers for Parallel Computing*, June 1998.

[12] Dean Tullsen, Jack Lo, Susan Eggers, and Henry Levy. Supporting fine-grain synchronization on a simultaneous multithreaded processor. In *5th International Symposium on High Performance Computer Architecture*, January 1999. Simultaneous Multithreading Home Page: http://www.cs.washington.edu/research/smt/index.html.