Automatic Parallelization for Java

Submitted by:
Mathieu Thibault-Marois – 5049388

Presented to
Professor Miodrag Bolic
for
ELG7199

Sunday April 29th, 2012
University of Ottawa
# Table of Contents

List of Figures ................................................................................................................................................ ii
List of Tables .................................................................................................................................................. ii

1. Introduction to Parallelization .................................................................................................................. 1
   1.1 Types of Dependences ........................................................................................................................ 1
      1.1.1 Data Dependences ....................................................................................................................... 1
      1.1.2 Control Dependences................................................................................................................... 5
   1.2 Dependence and Parallelization ......................................................................................................... 6
   1.3 Run-Time Approaches ......................................................................................................................... 8
      1.3.1 Speculation ................................................................................................................................... 8
      1.3.2 Inspector-Executor..................................................................................................................... 10

2. Parallelization in Java .............................................................................................................................. 11
   2.1 Early Work ......................................................................................................................................... 11
   2.2 Run-time Checks ............................................................................................................................... 12
   2.3 Thread Level Speculation .................................................................................................................. 13
      2.3.1 JavaSpMT ................................................................................................................................... 13
      2.3.2 Jrpm............................................................................................................................................ 15
   2.4 Method-Level Parallelism ................................................................................................................. 19
   2.5 Traces as Units of Parallelism ............................................................................................................ 20
      2.5.1 Bradel and Abdelrahman ........................................................................................................... 20
      2.5.2 Sun and Zhang ............................................................................................................................ 24

3. Targeting GPUs ........................................................................................................................................ 26
   3.1 Background ....................................................................................................................................... 26
   3.2 Static Parallelization ......................................................................................................................... 27
   3.3 Run-time Parallelization ................................................................................................................... 28

4. Overview of Covered Approaches .......................................................................................................... 34

5. Future Work and Conclusion .................................................................................................................. 35

References ..................................................................................................................................................... I
List of Figures

Figure 1: Data dependence example ................................................................. 1
Figure 2: Example of RAW, WAR and WAW data dependences ........................................ 1
Figure 3: Loop nesting example ........................................................................... 2
Figure 4: Distance and direction vector example .................................................... 4
Figure 5: Iteration space with dependences for statement S3 in Figure 4 ...................... 4
Figure 6: Branch types example ........................................................................... 6
Figure 7: Example of the impact of a loop-carried dependence on parallel execution .......... 7
Figure 8: Loop of Figure 7 parallelized on four processors without taking into account the loop-carried dependence ........................................................................ 7
Figure 9: Parallel program of Figure 8 with added locks to ensure proper execution order. 7
Figure 10: Example of a loop with a possible dependence between A and B due to aliasing .......................................................... 7
Figure 11: Data state used in [3] to track variable accesses and transitions .................. 9
Figure 12: JavaB parallelization flow. Taken from [7]. ............................................. 11
Figure 13: Structure of a program after creation of safe and alias-free regions with run-time checks. .12
Figure 14: Multidimensional arrays in Java implemented as arrays of arrays. Array A and B overlap. ... 13
Figure 15: JavaSpMT pipelined execution model. Taken from [9]. ............................. 14
Figure 16: Jrpm Block Diagram. Taken from [10] .................................................... 16
Figure 17: Hydra block diagram. TLS hardware is shown by the dotted lines, TEST hardware is shown in dark gray. Taken from [10] ................................................................. 16
Figure 18: Example of the execution of a TLS loop. Taken from [10] ......................... 18
Figure 19: Example of the packaging process for a trace covering basic block BB1, 2, 4, 5 and 6. Taken from [14]. ........................................ 22
Figure 20: Flow of the automatic parallelization process. Taken from [17]. ................. 24
Figure 21: General steps required by the execution of a CUDA kernel. Taken from [18]. .......................................................... 27
Figure 22: Process of executing a CUDA kernel from a JAVA program using JNI. Taken from [18] ........... 27
Figure 23: Block diagram of the proposed approach, including RapidMind. Note that the approach does not use the RapidMind front end and instead directly generates the IR code. Taken from [22]. .......................................................... 29
Figure 24: CPU code, on the left, transformed for GPU execution, on the right. The types of loops identified in the CPU code are shown. Taken from [22] ......................................................... 31

List of Tables

Table 1: Conditions that must be met for a loop to be considered for TLS [10] ................... 17
Table 2: Characteristics of different approaches to parallelizing Java programs on multicore CPUs .... 34
Table 3: Characteristics of different approaches to parallelizing Java programs on GPUs ........................ 34
1. Introduction to Parallelization

This section introduces some basic concepts of dependence and present why dependences pose a problem for the parallelization, or vectorization, of programs. It also introduces some generic frameworks for dealing with dependences at run-time.

1.1 Types of Dependences

When statements in programs read or write the same memory locations, their relative placement from each other can influence the outcome of operations based on the content of these same memory locations. These dependences must be taken into account when a program is modified as they must not be violated for the program to be kept equivalent, to have the same result[1]. They can be classified under two main types: data and control, or flow, dependences[1][2].

1.1.1 Data Dependences

Data dependences are created when at least two statements use data from the same memory location with at least one doing a writing operation[1]. Let us consider the very simple code in Figure 1 as an example: S1 and S2 need to be executed before S3 so that the result of S3 is accurate. If these two statements are moved after S3, then the program is no longer equivalent since it will not have the same behavior.

![Figure 1: Data dependence example](image1.png)

Data dependences can be further classified into three groups according to the order in which the read/write operations are made: read-after-write (RAW), write-after-read (WAR) and write-after-write (WAW)[2]. All three types are shown in Figure 2 below. There is a RAW dependence between S1 and S2 since S2 reads the value of X after S1 writes to it. Between S2 and S3 we have the opposite relation, WAR, since S3 writes the value of X after S2 reads it. And finally, there is a WAW dependence between S1 and S3 since both write to the same location and interchanging them would lead to a change in the value of Y in statement S2. Reordering any of the statement in this example would lead to a change in program behavior.

![Figure 2: Example of RAW, WAR and WAW data dependences](image2.png)

The notation for these dependences can differ from one source to another. In the work surveyed here[1][2], a RAW, or flow, dependence can be denoted as $s_x \delta_s y$ or $s_x \delta_s y$, a WAR, or anti, dependence can be denoted as $s_x \delta s_y$, and a WAW dependence as $s_x \delta^o_s y$.
The problem gets more complicated when loops are introduced since statements from different iterations can then have dependences between each other. While this might be considered a control dependence at first sight, it is not since only the way data dependences extends over loops iterations is considered, not if the loop will execute or not or when it will finish. To describe this situation, definitions for various concepts first need to be formalized[1]:

**Iteration Number:** The iteration number of an iteration of a loop is equal to the current loop index \(I\) minus the starting index \(L\) divided by the stride \(S\) of the loop plus one:
\[
(I - L)/S + 1.
\]

In the case where the loop starts at a value of 1 with a stride of 1, then the iteration number is equal to the index of the loop. Next, if the concept is pushed a little further, there is nothing to prevent a loop to be nested inside another loop. In that case, the iteration of the innermost loop must be defined with respect to the iterations of the outer loop[1]. In Figure 3 for example, the inner loop (index \(j\)), can take any iteration number values from 1 to 10. However, all 10 iterations are repeated for all values of index \(i\) for the outer loop.

```
S1 for(i=1,i<=10,i++){
S2   for(j=1,j<=10,j++){
S3     Z = i+j;
   }
}
```

*Figure 3: Loop nesting example*

To properly define an iteration of the inner loop, it must also be defined in which iteration of the outer loop it is executing. To do this, the concept of iteration vector is introduced[1]:

**Iteration Vector:** The iteration vector contains the iteration number of all loops from the outermost to the loop itself \((n)\):
\[
i = \{i_1, i_2, ..., i_n\}.
\]

For example, in Figure 3, the execution of S3 in the 3rd iteration of the inner loop and during the 2nd iteration of the outer loop would be noted as follow: \(i = \{2,3\}\). All possible iterations vectors for a particular statement are grouped together in what is called the iteration space[1]:

**Iteration Space:** Set containing all the possible iteration vectors for a particular statement.

Again referring to Figure 3, the iteration space of S3 is equal to:
\[
\{(1,1) \ldots (1,10)\} \left\{ \vdots \quad \vdots \right. \left\{ (10,1) \ldots (10,10) \right.\}
\]
Using the iteration vectors, it is possible to state if an iteration precedes another one or not[1]:

**Precedence:** Considering the following iterations vectors: $i = \{i_1, i_2, ..., i_n\}$ and $j = \{j_1, j_2, ..., j_n\}$, $i$ precedes $j$ if and only if:

\[
\begin{align*}
\{i_1, i_2, ..., i_{n-1}\} &< \{j_1, j_2, ..., j_{n-1}\} \text{ OR} \\
\{i_1, i_2, ..., i_{n-1}\} &> \{j_1, j_2, ..., j_{n-1}\} \text{ AND } \{i_n\} < \{j_n\}
\end{align*}
\]

In words, this means that if an iteration has a smaller iteration number for any of the outer loops, then it must execute before, whatever the iteration number for the inner loop. And in the case where the outer loops all have the same iteration number, then one must obviously compare the iteration number of the inner loop to know if an iteration precedes another. Equivalence between vectors can be defined similarly, simply requiring that all elements of both iteration vectors be equal in value. Using the concept of precedence, it is possible to define dependence[1]:

**Loop Dependence:** A dependence exists between two statements if:

1) Iteration $i$ precedes iteration $j$ or $i$ is equal to $j$ and there exist a path between the two statements
2) The 1$^{st}$ statement accesses location $M$ at iteration $i$
3) The 2$^{nd}$ statement accesses location $M$ at iteration $j$
4) At least one of the two accesses is a write operation.

The notion of dependence in the context of a loop has been defined, but the dependence itself has not been characterized. To do so, the concept of the distance vector, distance between sink and source of the dependence, and the concept of direction vector, direction in which the dependence flows, are defined here[1]:

**Distance Vector:** The distance vector $(\sigma = (\sigma_1, ..., \sigma_k))$ between an iteration represented by vector $i$ and an iteration represented by vector $j$ is defined by $\sigma = i - j$.

**Direction Vector:** The direction vector $(\rho = (\rho_1, ..., \rho_k))$ between two iteration vectors $i$ and $j$ is defined by the sign of the elements of the distance vector $(\sigma = (\sigma_1, ..., \sigma_k))$:

\[
\rho_n = \begin{cases} 
< & \text{if } \sigma_n > 0 \\
= & \text{if } \sigma_n = 0 \\
> & \text{if } \sigma_n < 0
\end{cases}, \quad \forall n \in N: 1 \leq n \leq k
\]
Figure 4 shows an example of how to apply these definitions to a concrete case.

| S1 | for(i=1,i<=4,i++){ |
| S2 |      for(j=1,j<=4,j++){ |
| S3 |          array[i+1][j-1] =array[i][j] |
|     | } |
|     | } |

Figure 4: Distance and direction vector example

Looking at the index function for the array in S3, one can state the following:

\[ i_1 + 1 = i_2 \Rightarrow i_2 - i_1 = 1 \]
\[ j_1 - 1 = j_2 \Rightarrow j_2 - j_1 = -1 \]

Then by using the definition of the distance vector, one can calculate the distance:

\[ \sigma = (i_2 - i_1, j_2 - j_1) = (1, -1) \]

This then gives the direction vector for the dependence:

\[ \rho = (<, >) \]

Graphically, the dependence over the iteration space for statement S3 can be represented as in Figure 5. Notice that the arrow representing the dependences cover the exact same distance as was described by the distance vector, always (1,-1).

Figure 5: Iteration space with dependences for statement S3 in Figure 4.

In order to parallelize a program, one must often apply some transformations that will help re-order and ease dependences, making it easier to split the workload across multiple parallel computing units. However, these transformations must respect basic rules in order for the program to remain equivalent to the sequential version[1]. For loop carried dependences, this can be defined as follows[1]:

---

Page | 4
**Direction Vector Transformation:** A transformation on a direction vector is valid if it does not make the left-most non-"=" element in any direction vector ">".

In words, this means that you can re-order loop iterations in any way as long as you do not change the order of the sink and source of a dependence. For example, if loop iteration 2 depends on loop iterations 1, it is not legal to reorder loop iteration 1 in such a way that it would execute after the 2nd iteration. The same observation can be done in the general case[1]:

**Theorem of Dependence:** Any transformation which simply changes the order in which statements in the program are executed and preserve the relative order in which the source and the sink of dependences are executed, also preserves the meaning of the program.

### 1.1.2 Control Dependences

Control dependences are created when the execution of a statement depends on the outcome of another[1]. This situation arises in the presence of forks, created by branching statements, in the program execution where the execution flow will either follow one path or the other. There are more than one way of dealing with control dependences. They can be removed through a process called if-conversion[1], or they can be integrated as a new type of dependence in the dependence graph[1]. The first option might seem very interesting at first since it is simply trying to remove control dependences but the technique requires the use of a predicable instruction set[1] so that "guards" that will be determining if an instruction will be executed or not can be evaluated at run-time. On top of that, since no analysis is made to determine if a piece of code is parallelizable or not before applying if-conversion, then predicated instructions might, and most likely will, be introduced needlessly[1]. For these reasons, this section will concentrate on integrating control dependences in the dependence graph by characterizing them.

As said before, control dependences are created at forks in the execution flow, at branch statements. They can be further classified according to the direction in which the flow is diverged[1]. A forward branch transfers the flow to a statement that occurs after the branch but at the same nesting level[1]. A backward branch transfers the control flow to a statement that occurs before the branch, still at the same nesting level[1]. An exit branch transfers the flow of control to a statement that is located outside the loop nest and thus terminates the loop[1]. Finally, there is the "enter" branch type, which transfers the flow of control from outside the loop nest to inside the loop nest[1]. While these are possible in some languages, they are often against common programming practices since they often result in code spaghettification and hard to understand programs.
Figure 6 shows the three main types of branches. Statement S2 transfers control to statement S3 or S4, and is of the forward type since these execute after and in the same loop nest, statement S5 transfers control to S1, which executes before, or exits the loop and is thus both of the backward and exit type.

Formally, control dependences can be defined as follow[1]:

**Control Dependence:** A control dependence from a statement x to a statement y exist if
1) A path from x to y exist,
2) Every statement other than x in between x and y on the path is post dominated by y, and
3) x is not post dominated by y.

**Post Dominance:** If, in order to reach the exit node from a node x, all paths from x to the exit node must pass through a node y, then x post dominates y.

### 1.2 Dependence and Parallelization

Now that dependence has been defined, one might wonder about how, concretely, dependences influence the process of parallelizing a program. Note that the goal of this section is not to introduce theory on how to test for dependence or to explain transformations that helps eliminating these dependences; it is simply to put in perspective that testing is difficult and that determining relations accurately is essential to correct parallelization. First of all, let us go through an example to explain why parallelizing a loop with a loop-carried dependence causes problem.

![Figure 7: Example of the impact of a loop-carried dependence on parallel execution.](image)

The loop illustrated in Figure 7 carries a RAW dependence of distance 1: iteration i+1 writes to the same memory space that is read by iteration i. Now let us suppose that array A has the value shown in Figure 7 and that the loop is parallelized on a four processor system as shown in Figure 8.
If the program executes in the order CPU1->CPU2->CPU3->CPU4, the result will be the same as the one shown in Figure 7. But if, for example, the CPUs finish their work in the order CPU4->CPU3->CPU2->CPU1, the result will instead be $A = \{10,10,10,10,10\}$. This simple example illustrate why blindly parallelizing a loop without taking into consideration dependences is not possible if the program is to preserve its meaning.

A simple solution that might come to mind at this stage is that the programmer could add locks to ensure that the program executes in the proper order. The parallelized program of Figure 8 with added synchronization is shown in Figure 9.

While this program does indeed always execute in the proper order, it forces each CPU to wait for the previous CPU to finish its execution and thus results in a serialization of the writes operation. The result is that this parallel program executes sequentially. Thus, blindly synchronizing is not a solution either. This example is overly simple and does not represent a typical case that would be considered in parallelization, but it does highlight some problems with the obvious approaches.

Testing for dependence is the process of trying to disprove the existence of dependences [1]. When the analysis cannot disprove a dependence, it must conservatively assume that it exists. This is to prevent changes to the meaning of the sequential program.

Some language features can make it considerably harder to analyze dependences. For example, in Java, the two arrays in Figure 10 may be aliases of the same array, that is the reference to array $A$ and $B$ may point to the same memory space at run-time. An off-line analysis must thus assume that a dependence between these two arrays can exist since it does not have the information necessary to disprove its existence. Even worse, in C++, an array can reference an arbitrary element of another array, thus even though the two references will not be equal at run-time, the memory space they use overlaps[1].

```
S1  for(i=0;i<4;i++)
    {                        
      A[i]=B[i+1]
    }
```

Figure 10: Example of a loop with a possible dependence between $A$ and $B$ due to aliasing.
Even when simply considering array indexes without the previously mentioned issues, the analysis can easily become non-trivial. In the general case, where index expression can take any forms, testing for dependence is currently considered an undecidable problem[1]. Even if index expressions are limited to be linear functions of the loop index variables inside a loop nest, determining dependences is NP-Complete[1]

1.3 Run-Time Approaches

The discussion so far covered what is dependence and how can we test for it. However these assumed that the necessary knowledge to do so was available at compile-time while in most cases, it is not and it is thus impossible to disprove the existence of dependence at compile-time. To deal with this problem and to allow further exploitation of parallelism, run-time approaches are used.

1.3.1 Speculation

Speculation may seem like an obvious solution: if you cannot disprove the dependence, assume it does not exist and speculatively execute the piece of code in parallel. In the event that the dependence does occur, detect and deal with the problem at run-time. While this is indeed the main idea behind speculative execution, it will become clear during the reading of the present section that, in reality, the situation is much more complex than it first seems.

Data dependences must already be identified for the process of speculation to begin[3]. Speculation can be coupled with any compile-time dependence testing and, in fact, doing so might eliminate dependences and allow for more speculative parallelism to arise[4]. The process is independent of whether or not the code was previously transformed [3].

The first task is to identify where speculation will take place and to modify the code so that it is possible to keep track of which thread accessed what data and in what way[3]. To do so, load and stores to data that is classified as being speculative must be modified so that they also update the access data structure[3]. This is done by adding new instructions that ensure that the updating process is done appropriately [3]. To know the exact sequence of instructions, the reader is directed to [3].

Since multiple threads might be writing/reading the speculative data, multiple copies must be kept so that it is possible, in the case of a violation, to go back to the correct value[3]. This means that there are not only multiple copies of the data, one for each active thread, but also multiple possible states for each of the different version of the data[3]. Typical states for the speculative data are[3]: not accessed, modified, exposed loaded, and exposed loaded but later modified. An exposed load is when the first operation on a speculative piece of data by a thread is a load, with no store operation having been done before it[3]. Remember that these states are per thread and that each thread may have a different state[3].
Figure 11: Data state used in [3] to track variable accesses and transitions.

Once the speculative section has executed without any violation, the data can then be considered safe and can be committed [3]. Different schemes are possible: data can be committed along the way, as each thread safely finishes its speculative section, or it can be committed only once, when all threads are finished [3]. Committing along the way can help to free memory space by removing the thread copy and its associated speculative access structure [3].

Once the loop has been partitioned into several chunks to be executed speculatively, these parts can then be scheduled on the available processors [3]. If the number of chunks is equal to the number of processors, the scheduling is done statically [3]. On the other hand, in the case where the number of chunks is larger than the total number of processors, then the scheduling must be done dynamically [3] as the chunks finish their execution as there are still more threads to serve.

The final task to be performed during speculative execution is the termination of threads and the subsequent restarting of the execution upon the detection of a dependence violation [3]. Using the speculative access structure, it is possible to determine if a violation is occurring [3]. For example, if a thread i is executing later iterations of the loop and does a read while another thread j that is executing earlier iterations of the loop has the same variable in modified state, then i is actually reading the wrong value and is violating a dependence and must be squashed and restarted [3].

Not only the thread must be restarted, but it must be done in such a way that this dependence is not violated again [3]. There are multiple ways to ensure that the thread will proceed correctly the next time: synchronization instructions can be added to the violating instruction so that it does not proceed until the value it needs is available, or the restarting process can simply wait that all threads executing prior iterations are finished before re-launching the offending thread [3].
The frequency of data violation checks are also left to the specific implementation[3]. For example, data dependence violation can be checked up at every load and store, when the thread tries to commit, or at the end of the speculative section[3]. Increasing the frequency of violation checks increases the overhead but allows the execution to be stopped sooner and prevents wasting time on useless computations[3]. On the other hand, a lowering check frequency lowers the overhead introduced by checkups but results in more computation being done uselessly between the time of the violation and the time of the verification[3].

Some squashing instances can be prevented by allowing the forwarding of data between threads[3]. For example, upon detection of a read on data that is modified in a thread running preceding iterations, instead of squashing the thread, it might be possible to ask the preceding thread for its updated copy of the data and continue executing with this value[3].

Obviously, all these tasks that need to be performed during speculative execution introduce a non-negligible amount of computation and storage overhead[3]. A loop must offer enough parallelism to overcome this overhead to make its parallelization a viable option. In order to prevent excessive squashing, any speculative scheme should provide some means of deciding if a speculative execution should proceed further or stop and revert back to the sequential code.

1.3.2 Inspector-Executor

The speculative approach assumes that dependences will not exist at run-time to parallelize the loops and then deals with the consequences if they do exist in the end. The inspector-executor scheme instead tries to prove that dependences do not exist once the necessary data is available at run-time and execute the loop in parallel if all the dependences are disproved[5]. The advantage is that once the decision is made to run the loop in parallel, it is known that it can be done and that no violation will arise[5]. The disadvantage is that the analysis on the dependences can be quite long to complete[5].

Inspector-executor is merely a framework for parallelization[5]. It does not specify in itself how data dependences should be tested or how parallel threads should be run. There exist many different configurations of inspector/executors in the literature[5]: a scheduling inspector with an executor, an inspector with a scheduling executor and an inspector, a separate scheduler and an executor. The data dependence analysis ran by the inspector can, in itself, be parallelized and doing so helps reduce the associated overhead[5].

Some parts of the dependence analysis can run directly in the executor instead of in the inspector [5]. However, this approach is not recommended since the executor will not retain this information for future loop execution while the inspector will, reducing the time to analyze subsequent executions of the same loop[5].
2. Parallelization in Java

For certain languages, automatic parallelization is a fairly old topic. For example, Paraphrase, an automatic parallel compiler for FORTRAN program, was developed in 1981[6]. By comparison, the first attempt at an automatic compiler for Java dates back to the late nineties[7]. This section will present different approaches to parallelizing Java programs starting from early off-line compilers and moving to newer, run-time techniques.

2.1 Early Work

One of the earliest work on parallelization for Java is the JavaB compiler[7], first presented in 1998. It is a static analysis tool, written in C, which works on Java byte-code to identify implicit loop-level parallelism[7]. An advantage of working with byte-code is that the source code need not be available at the time the program is transformed[7].

![JavaB parallelization flow. Taken from [7.]](image)

JavaB does the dependences analysis and transforms parallelizable loops using the multithreading mechanisms that are available in the JVM[7]. This ensures that the parallelized version can still be executed on any JVM[7]. The output is also still in byte code, in order to retain the portability of the original [7]. The sequential loop is replaced by a piece of code that initializes the threads that will run the parallel version of it[7].

In order to help eliminate the most dependences possible, JavaB also supports an interactive mode where, when unsure about whether or not a dependence exist, it will ask the user if the dependence exists or not[7]. This is an interesting feature but since there may be a vast number of these dependences that cannot be disproved, this approach can put a non-negligible burden on the user[7]. This is especially true when the user does not have access to the source code and can only rely on the byte code for interpretation[7]. Not only that, but it also opens the door for user errors which can then lead to a parallelized version that is non-equivalent and might even be unstable. The compiler can also be used merely as an analysis tool and be told not to produce a new class file[7]. In that case, the compiler will simply do its dependence analysis and report its conclusion.
2.2 Run-time Checks

Like it was mentioned in early parts of the report, many dependence relations cannot be disproved until runtime, limiting the effectiveness of static analysis techniques. The work of Artigas et al. in [8] resembles the previous discussed inspector/executor scheme in that it assumes that some dependences will not be present at compile time to create a second version of the loop on which more optimization can take place and then, at run-time, dependences are verified and the version to be executed is chosen accordingly [8]. The authors identify two main sources of dependence in Java that prevents the use of extensive code transformations: exceptions, such as null-pointer checks and index out of bound checks, and aliases, for loops with more than one array being accessed [8].

First, let us consider the problem of exceptions. Exception checks for bounds or null pointers prevent any reordering of array references in Java [8], thus to enable transformations to take place, these must be speculated not to be thrown at run-time [8]. To do so, the compiler creates a “safe” region, which is a copy of the original, unsafe region with exception checks edited out [8]. Further optimization can then be done on this new region. Tests are then added to decide at run-time whether or not exceptions can be thrown and, depending on the result, execute the safe or unsafe version [8].

For the case in which multiple arrays that may alias to each others are accessed in the region, the procedure is similar except that it is carried out after the creation of the safe region since removing aliases on the unsafe region would yield no benefits anyway since the exceptions are preventing the reordering of instructions [8]. A copy of the safe region created previously is thus made and a new region, in which the compiler speculates that references will not alias at run-time, is created [8]. Further checks are added to test if whether or not the assumption holds at run-time. If it does, the alias-free version is executed, otherwise the simpler safe version executes [8]. This is assuming that the region has been previously declared safe, otherwise alias checks need not be done and the unsafe version is chosen. With the possibility of aliases removed, the alias-free version can be further optimized compared to the safe region. Figure 13 shows the structure of a program after the creation of both a safe and an alias-free region.

```java
if(*Bounds and null checks do not hold* == true)
{
    if(*alias checks do not hold* == true)
    {
        //Alias-free and safe region
    }
    else
    {
        //Safe region
    }
}
else
{
    //Unsafe region
}
```

Figure 13: Structure of a program after creation of safe and alias-free regions with run-time checks.
Testing if two arrays alias is simple for the one-dimensional case in Java since the language does not allow an array to point to arbitrary elements of another[8], like in C++ for example, and thus to check if two arrays overlap is as simple as comparing their references. The multidimensional case is much more complicated however since Java implements multidimensional arrays as arrays of arrays. Figure 14 illustrate the problem that arises due this situation: both A[1] and B[1] overlap despite A and B having distinct references.

![Figure 14: Multidimensional arrays in Java implemented as arrays of arrays. Array A and B overlap.](image)

Considering only distinct arrays, there are five in Figure 14 for example, being written to, Nw, or referenced, Nwr, testing if any of these operations target the same memory location will in the worst case require $O(N_w \times N_{w_r})$ reference comparisons[8]. Multidimensional arrays are detrimental for the performance since they greatly increase the total time spent on testing for aliases[8]. In such a case, it might be better to ignore the alias-free region and simply stop at creating the safe region.

2.3 Thread Level Speculation

2.3.1 JavaSpMT

JavaSpMT[9] is similar in many ways to the preceding approach: it was introduced on the same year, 2000, it performs transformations at the source level to maintain portability[9], it is able to perform run-time dependence checking[9] and it uses the Java thread model to implement parallelism[9]. The comparison, however, stops there. JavaSpMT uses a pipelined execution model and uses a speculative approach that can deal with control dependences and a wide variety of loop structures[9]. Although it is implemented on a shared memory system, JavaSpMT is implemented purely in software and does not require any specific hardware configuration by design[9]. Support for forwarding data in a system without shared memory could be added for example[9].

The pipelined execution model used by JavaSpMT has a total of four stages[9]: continuation, target-store-address-generation (TSAG), computation and write-back. In the first stage, the thread computes the variables, such as loop index, needed to fork the next thread[9]. When the thread finishes this job, it starts the next thread, forward the needed data and moves to the second stage[9]. In the TSAG stage, the thread computes the target addresses of all write operation upon which the successor may be dependent[9]. Once done, it forwards this data to the successor thread, set a flag to notify the completion of the stage and proceeds to the next[9]. The successor thread uses the flag as
asynchronization variable and cannot enter the TSAG stage unless the predecessor has finished[9]. Computing these addresses allows the run-time checking of dependences by allowing threads to know whether or not load and store are done at the same memory location or not[9]. The computation stage contains the actual computations to be done by the thread[9]. This is where the actual data is derived and stored. Data that was determined as being dependent must be forwarded to the successor. When the successor reaches an instruction for which it needs data computed by its predecessor, it looks into its memory buffer for the data and reads it from there if available and otherwise wait for the predecessor to forward the data[9]. A successor can actually finish its computation stage before the thread that spawned it in the case where no data dependencies are hindering progress, but it cannot enter the write back stage until the predecessor has finished writing back itself[9]. This ensures that threads will be retiring in order, preserving the program order, and that speculative threads verify their execution was not aborted before committing their data[9]. In the case of the shared memory system used, the write-back stage is actually empty for non-speculative threads since they directly update the shared memory[9]. In systems where memory is private to each processing units, even non-speculative thread would have to commit their data before retiring to ensure that the global data is kept up to date[9].

Speculative threads in JavaSpMT are only created in the case of a control dependence that cannot be decided upon at the current time[9]. The framework thus spawns threads to deal with all of the possible execution flows and, when the control data is available, abort all speculative threads for paths that were wrongly executed[9]. Of course, if a thread is aborted, all of its successors must also be
aborted since their creation should not have happened in the first place[9]. Each speculative thread has an abort flag that it is set by the main thread that spawned it[9]. Each successor checks this flag on two different occasions[9]: at the start of the computation stage and at the start of the write back stage. Since a thread cannot enter write-back before all its predecessors are themselves finished writing back, a speculative thread is thus certain that its execution proceeded correctly before any data is committed[9]. Reading the abort flag before the computation phase allows the thread to bypass that stage and directly abort, without wasting time on computations that will anyways be useless[9]. When a thread reads its abort flag and returns true, it must also set the abort flag true for all its successors before aborting itself[9].

Managing the pipeline stages, the data communication between threads and the creation of threads at run-time obviously introduces some overhead that needs to be measured. Since threads are created using the Java thread package, the associated overhead is dependent on the implementation of this package for the specific platform[9]. For the management of pipeline stages, a fixed number of library calls are required per stage[9]. This part of the overhead is thus bound by the worst case, which is when a thread goes through all stages before finishing[9]. The communication overhead is more complex to bound as it depends on architecture details and the average number of communications a thread needs to do with other threads to forward addresses or data, set flags and commit data[9]. Due to the shared memory architecture used in[9], the communication time is negligible since all reads/writes are done directly to the variable in the shared space. The only time some kind of communication is still necessary is when the local updates done by a speculative thread need to be transmitted to the shared memory[9].

Comparison of performance was done versus the JavaB parallelizing compiler[7], which was discussed in the early work section of the present document. For a matrix multiplication algorithm and varying matrix sizes, JavaB was found to be 2-4% faster than JavaSpMT on average[9]. This situation can be attributed to the overhead associated with the run-time approach versus the static, off-line, analysis of JavaB. While JavaSpMT is slower in this particular case, it is able to deal with situations that JavaB cannot, such as speculating on control dependences and parallelizing do-while loops[9]. JavaSpMT is thus applicable to a wider range of applications than JavaB and can hope to improve the performance of programs that would be impossible to analyze using a static approach.

2.3.2 Jrpm

One problem with speculation is that the analysis to determine which threads should be speculatively executed is usually done at compile-time. By doing the analysis at run-time, it would be possible to collect data on a particular piece of code, for example the average number of iterations or the frequency at which dependences are violated, to determine which pieces of code have the best potential for parallelization. This would prevent the blind execution of speculative versions of loops that present poor parallelism potential at run-time. This is the approach taken by the Jrpm system[10]. It uses a chip multi-processor (CMP) with hardware thread-level speculation (TLS) support, Hydra, along with a hardware profiler, Tracer for Extracting Speculative Threads (TEST)[11], and a just-in-time (JIT) compilation capable virtual machine, Kaffe with microJIT, to do run-time analysis of sequential code,
determine which loops are suitable for speculation, run-time re-compilation into a parallelized version, and run-time enforcement of data dependences[10].

Figure 16: Jrpm Block Diagram. Taken from[10].

Figure 16 shows the flow of execution of the Jrpm approach. Java bytecode is first compiled by the VM and some instructions are added to allow the hardware to keep track of execution and collect statistics[10]. The annotated program is then run on the Hydra CMP, data is collected and fed back to the analyzer inside the VM[10]. The profiler decides which loops have the potential for the best speed-up and instructs the JIT the recompile these using multiple threads[10]. The next time this piece of code is ran, the speculative version will execute instead of the sequential one and consistency is enforced by the hardware support for TLS inside Hydra[10].

Figure 17: Hydra block diagram. TLS hardware is shown by the dotted lines, TEST hardware is shown in dark gray. Taken from [10].
The Hydra CMP, shown in Figure 17 along with the additional TEST hardware, is, at the base, a typical 4 cores single-issue, pipelined MIPS multiprocessor[10]. What makes this processor special is the additional hardware support for speculative thread execution: all processors are accompanied by a co-processor which manages thread speculation, data caches are extended with speculative tag bits to detect inter-thread dependence violations and additional buffer space is available to store speculative data until it is either committed or discarded[10]. The TLS hardware guarantees that the memory ordering of the original sequential code will be preserved[10]. By forwarding data to threads executing later iterations of the parallel loop and by restarting threads that read values that are not ready yet, RAW dependences are enforced[10]. The buffering of speculative data and the commit process ensure that the ordering of writes operations is preserved, ensuring that no WAW violations will happen[10]. Since buffered data is only available to successor threads, WAR violations simply cannot happen[10]. The additional hardware is managed through special TLS instructions in the Hydra’s instruction set[10].

The TEST hardware[11] collects the information necessary during the execution of the sequential code to determine which loops have the potential to be parallelized[10]. This includes information about dependency timing, thread length, and estimates of the buffer space that would be used by a speculative version[10]. The analysis on buffer space is necessary to ensure that a parallelized version of the loop will fit inside the buffer available or otherwise the execution will stall and performance will decrease dramatically[10]. Speculative loops are executed as one thread per iteration in Jrpm[10]. Thus, when analyzing sequential loops, the TEST hardware can interpret all iterations as tentative speculative threads, and analyze dependence between iterations as being dependences between threads[10]. The sequential loop will be analyzed until a conclusion can be drawn on its parallelizability. In the covered paper, this means at least a thousand iterations have been executed[10]. Then, if certain conditions are met, shown in Table 1, the loop will be recompiled speculatively by the JIT compiler for future use[10]. A loop which has been parallelized is called a speculative thread loop (STL)[10].

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average iterations per entry</td>
<td>&gt;&gt; 1</td>
</tr>
<tr>
<td>Speculative buffer overflow frequency</td>
<td>&lt;&lt; 1</td>
</tr>
<tr>
<td>Predicted speedup</td>
<td>&gt; 1.2</td>
</tr>
</tbody>
</table>

The use of a hardware analyzer is justified by the fact that simulations have shown that a software implementation would be over 100 times slower and that the transistor count overhead is less than 1%[10]. Even though the analysis process is much faster in hardware than in software, the overhead is still present and attains non-negligible levels: 7.8% of the execution time on average for the set of benchmarks considered in[10] and up to 25% in the worst case[10]. Even though the creators of Jrpm argue that a software solution is slow, a software implementation remains an open topic for other researchers[12].
Figure 18: Example of the execution of a TLS loop. Taken from [10].

Figure 18 shows an example of how the execution of a STL proceeds [10]. First, the master CPU starts the speculation process by clearing the store buffers, setting speculative event handlers, waking up slave CPUs, initializing thread level speculation and then begins the creation of new threads. When a thread reaches the exit condition, it waits till the CPU it is running on becomes the head CPU, commits its data and returns control to the master CPU. In the case of RAW violations, buffers must be cleared and the execution of the thread and its successors is restarted [10].

The JIT compiler is able to perform a number of optimizations that can help improve the performance of STLs [10]. These transformations are performed automatically upon the recompilation of sequential loop to its speculative version [10].

Loop invariant registers are typically used to avoid having to needlessly reload values that will anyway never change every time they are read in a loop. To enable to same kind of allocation in Jrpm for STLs, the fact that a thread may be restarted must be considered as the invariant values will have to be restored [10]. An option would be to keep a shadow copy of the registers but this consumes precious space in the chip [10]. Instead, the relevant registers are pushed onto the stack and restored if the thread is restarted [10]. Thus, the compiler must not only add the allocation of the value to the register, but also the relevant stack operations [10].

Loop inductor variable result in a loop-carried dependency that becomes an inter-thread dependency in a TLS. Communicating the value of the induction variable from one thread to another results in the serialization of the execution and may cause RAW violations [10]. Instead, a hardware counter keeps track of the value of the iteration count and iterations are distributed in a round-robin
fashion[10]. To support this, the compiler must add, in the restart phase, code to recover the proper iteration number upon a thread restart[10]. In the case where the loop inductor can sometime take an unpredictable value, code must also be added to write the correct iteration value to the counter and to restart execution of the thread and its successors[10].

In the case where a certain dependency occurs often enough, it might be profitable to synchronize its access instead of possibly squashing threads[10]. In the case where TEST determines that a dependency is shown to occur more than 80% of the time, it will replace load operations to this location by a non-violating variant which implements a lock on which threads must wait before consuming the data[10].

Like most parallelizing compilers, microJIT can perform the reduction of operations such as computing sums[10]. The compiler instead makes each thread calculate an intermediate value to a local variable and the final value is computed at the end of the speculative loop.

In the case in which a parallelizable loop contains another loop, also parallelizable, which rarely executes, the compiler can insert handlers in the code so that when the inner loop executes, the current STL is switched to the inner loop and restored after[10]. This provides better load balance compared to if the outer-loop was parallelized but not the inner one since some threads will end up executing the inner loop while others won’t[10].

More than half of the execution time of the startup and shutdown phases is spent waking up CPUs for parallel work and initializing the hardware[10]. This phase of both startup and shutdown need not be repeated when multiple STLs are executed one after the other, only the management of the buffers need to happen before and after every STL[10]. The compiler can detect when a loop nest or a method is comprised of more than one STL and will eliminate the repetition by moving these phases to the start and end of the block[10]. This amortizes the overhead over multiple STLs.

The Jrpm approach was tested on a wide variety of benchmark, from typical synthetic benchmarks to real applications[10]. The conclusion is that speed-up ranges from 1.5 to 4 depending on the type of calculations made, with no benchmarks resulting in a decrease in performance when using Jrpm[10].

2.4 Method-Level Parallelism

While most of the work in automatic parallelization targets loops parallelism, the potential does not stop there. The zJava system[13] instead explores method-level parallelism by executing methods on different asynchronous threads. At compile-time, methods are analyzed to determine the data that they access and a description of these accesses is built[13]. Then, at run-time, when the method is called, the description is used to determine if dependences exist[13].
The execution of the program starts sequentially with the main method and then, at every method call, the description is checked and for every data access that is indeed dependent, a synchronization event is created[13]. Finally, an asynchronous thread is created for the execution of the method[13]. A method finishes when its thread terminates and the program finishes when all threads terminates[13].

The potential of such a system mainly lies in the fact that it is orthogonal to the loop parallelization approach and could be somewhat easily used complimentarily to further increase the amount of parallelism.

2.5 Traces as Units of Parallelism

A fairly recent approach, (the oldest work covered here, [14], dates back to 2007) in Java parallelization is the use of traces as the unit of parallelization instead of loops or methods. A trace is a sequence of basic blocks that corresponds to a path in the control flow graph and thus can cover both loop iterations and different methods[14]. This means that trace-level parallelism incorporates both data and task level parallelism[14]. The use of traces as a unit also does not preclude a particular approach to parallelization[14]. Trace-level parallelization can be done conservatively, waiting for dependences to become available, as well as speculatively.

Since the use of traces as a unit of parallelization in Java program is kind of new, studies had to be conducted to determine if Java applications offer enough trace-level parallelism for the approach to be viable[15]. The conclusion is that Java applications do indeed have enough trace-level parallelism for around 4 to 8 processors[15].

2.5.1 Bradel and Abdelrahman

The proposed execution model starts as follow[14]: a program starts executing sequentially, a monitor analyzes the program, detects frequently used parts and save traces associated with these parts. When a saved trace is reached, it is assigned to its own thread, that is a new thread is created for the trace, and then the most likely trace to succeed it is identified and also assigned to its own thread and so on until there are no more successors available[14]. Then the main thread waits for the execution of all the traces to finish and restarts execution sequentially until the next saved trace is reached[14]. A scheduler is responsible of dispatching threads executing traces on the different processors and to make sure that no dependence violation occurs between threads[14]. The responsibility of such a scheduler will be different depending on the parallelization approach taken. For example, if a conservative approach is used, then the scheduler will make sure that traces cannot proceed until the data they depend on is ready, but under a speculative approach, the responsibility of the scheduler becomes checking for dependence violations and aborting offending threads. In the first version of the presented work[14], dependences are maintained in software through the use of synchronization instructions; an extended version of the framework[16], which can handle recursive methods, uses hardware transactional memory.

Dealing with traces comes with its own set of non-trivial challenges. In order to decide which traces are collected and saved, there must be some criteria to decide which ones are good and which
ones are useless\cite{14}. The selection of this criterion must be done carefully as the traces that are saved are the only one that can be used to extract parallelism so if a criteria is too selective, it might end up limiting the amount of available parallelism\cite{14}. A criterion should also be simple enough as to incur only a small overhead\cite{14}. As mentioned before, when a trace is reached, it is assigned to a thread and the most likely successor is identified. The process of choosing a successor is non-trivial as knowing exactly which trace will follow requires the knowledge of all related control flow dependence value\cite{14}. Since this is often impossible to obtain, some heuristic will need to be applied to select a most likely candidate\cite{14}. Scheduling the traces on the system is also quite complicated. While the problem is very similar to the problem of scheduling threads containing loop iterations, traces are less structured and result in more complex load balancing and data locality analysis\cite{14}. Dealing with the dependences between threads is always a problem in parallel programs. As was mentioned earlier, there exist two possible solutions: either the system is implemented in such a way that threads wait for the data to become available or these are executed speculatively and are reset upon a violation.

When traces are deemed worthy of being saved, they must be packaged in such a way that they can be dispatched within a thread to a processor\cite{14}. The transformation process begins by identifying which basic blocks are part of the trace and what are the possible exit points of the trace\cite{14}. The second step is to create a new method that contains the basic block of the trace\cite{14}. Next, a basic block must be added for the entry and one for all possible exit points\cite{14}. Back in the original method, the entry block is then replaced by a block containing a call to the new method entry block\cite{14}. A jump is placed in all exit blocks in the new method to return to the original method\cite{14}. An added return block in the original method uses the return value to know which exit was taken\cite{14}. Finally, instructions are added in both methods to pass and restore variables between the two upon entry and exit\cite{14}. Figure 19, on page 22, illustrate this process through an example. The trace being considered covers basic block 1, 2, 4, 5 and 6 and the goal is to package it so that it can be dispatched as an individual thread. The basic blocks forming the trace are copied to the new method, basic blocks have been added to all entry and exit points, BB1 in the original method has been replaced with a call to the new method and jumps have been placed at the new method exit points to jump back to the return block in the original. Finally, instructions have been added to transfer and restore variables a, b and c where appropriate. The new method is now capable of being dispatched as a separate thread.

The compilation process has to be modified a little since a JIT compiler typically compile methods one at a time and thus cannot compile both the original and the new method at the same time\cite{14}. Creating the original first is impossible since there needs to be a call to the new method, which does not exist yet, and creating the new one first is also impossible since it returns back to the original method which has not been compiled yet\cite{14}. In order to get around the problem, the authors suggest the compilation of the original method first while replacing the call to the new methods to what they call a lazy compilation method\cite{14}. When this compilation method is called, it starts the compilation process for the new method and replaces the calls to point to the new method\cite{14}. 
Tasks are groups of traces that will execute in parallel[16]. These are formed to balance the amount of parallelism available and the overhead associated with parallelization[16]. Large tasks can contain a large number of instructions and since the parallelization overhead occurs only once, the overhead is appropriately spread over all these instructions, reducing the percentage of execution time it uses[16]. However, grouping many instructions in one task reduces the total amount of tasks in the program and may end up reducing the amount of available parallelism[16]. Of course, for two tasks to execute in parallel without worrying about violating dependences, control flow between the tasks would have to be known prior to the scheduling of these tasks. Since this is not always possible, tasks should at least be created in such a way as to maximize the predictability of the control flows between them[16]. The authors make the argument that a good choice to get this property is to have task boundaries created along cycles so that a task begins at the start of a cycle and finish at the end[16]. In a traditional loop, a cycle is created when, at the end of the loop, the control flow is redirected at the start of the loop[16]. Cycles can also be created as the result of recursion.

In order to perform the analysis necessary to the formation of tasks, a graph of traces is first created in which nodes are traces and edges are control flow between traces[16]. Then this graph is further divided into strongly connected components (SCCs), which are sub graphs in which there exist a path between any pair of nodes[16]. The result of this operation is that all cycles in the trace graph are encompassed inside SCCs and that no cycles exist between SCCs[16]. Since tasks boundaries should be created along cycles, this is the side effect that a task must be contained within an SCC[16]. A benefit
that arises due to this characteristic is the fact that the analysis can be partitioned to analyze each SCC separately instead of having to go threat to whole graph at once[16].

Edges in the SCC are categorized into three groups[16]: forward, backward and implicit edges. Forward edges represent a forward flow control, for example a jump to a later instruction, and also include control flow from return calls[16]. Forward edges are not considered during analysis and returns are put into this category in order to distinguish them from implicit edges[16]. Implicit edges exist between a call instruction and the instruction that will execute once this call finishes and execution returns to the current method[16]. Finally backward edges represent backward control flow[16]. The three types are defined formally as follow, with the edges set being control flow arcs in an SCC[16]:

\[
\begin{align*}
\text{Backward Edges:} & \quad BE = \left\{ e \mid e \in \text{edges}: e.\text{target} \leq e.\text{source} \text{ and } \text{return}(e) = \text{false} \right\} \\
\text{Forward Edges:} & \quad FE = \left\{ e \mid e \in \text{edges}: e.\text{target} > e.\text{source} \text{ or } \text{return}(e) = \text{true} \right\} \\
\text{Implicit Edges:} & \quad IE = \{ (e.\text{source}, t) \mid e \in BE \text{ and } \text{implicitedges}(e) = t \}
\end{align*}
\]

\[\text{Return}(e): \quad \text{Return true if e is based on a return statement or false otherwise.}\]

\[\text{Implicitedges}(e): \quad \text{If e is based on a call statement, returns the trace that starts after the call. If e is not based on a call statement, or if there is no trace, false is returned.}\]

Once the set of edges are created, it is then possible to identify starting, ending and forking points of tasks. First, the start point of a task should be at the target of a backward edge since this is the “start” of a cycle[16]:

\[\begin{align*}
\text{Task-start:} & \quad TS = \{ t \mid \exists e \in BE: e.\text{target} = t \} \\
\text{Task-end:} & \quad TE = \{ \text{lasttrace}(\text{traces}) \right\} \cup \{ e.\text{source} \mid e \in BE \text{ and } \forall f \in IE: f.\text{source} \neq e.\text{source} \}
\end{align*}\]

\[\text{Lasttrace}(\text{traces}): \quad \text{Returns the last trace of the set of traces given as input. For the purpose of the task forming algorithm the set of traces passed as input is the set of all traces inside the SCC being considered.}\]

Finally, task forks are identified by edges that point to the start of another task[16]. Obviously, these can only be identified after the task-start set has been created.

\[\begin{align*}
\text{Task-fork:} & \quad TF = \{ e \mid e \in \text{edges}: e.\text{target} \in TS \}
\end{align*}\]
2.5.2 Sun and Zhang

Apart from the work done by Bradel[14][15][16], the use of traces in the parallelization of Java programs was also investigated by Yu Sun and Wei Zhang in[17]. While similar to the original work done in[14], several differences are worth exploring.

Instead of collecting traces through preliminary runs of a program, they are collected at runtime[17]. First, filtering is done to detect so called hot methods, which are then instrumented to collect trace information[17]. This information is then passed through a cost/benefit model to determine if the method is a good candidate for parallelization[17]. While traces are used, only loop parallelism is targeted[17]. The flow of the approach is illustrated in Figure 20.

![Figure 20: Flow of the automatic parallelization process. Taken from[17].](image_url)

Two types of traces are collected at run-time[17]: the execution trace, which contains the sequence of basic blocks executed, this is the type that was referred by the term trace in the preceding section, and memory access trace, which contain the sequence of read and writes operation. The latter is useful to detect dependences between traces. The collection of the memory information was implied in the all the run-time approaches discussed so far as the information is necessary for an accurate analysis of dependence. The collection of traces must balance accuracy and overhead. Collecting accurate information takes time and not collecting enough information reduces the available parallelism. To help reduce overhead while maintaining the accuracy of information, the approach proposes the uses of different details levels for different parts of the program[17]. Since the approach targets loop-level parallelism, loops will be the target of the highest level of precision while the rest of the program will be covered with less accuracy[17]. Thus for loops in hot methods, both memory access trace and
execution trace are collected[17]. For the non-loop parts of hot methods, collecting only the execution trace is deemed sufficient[17]. For methods not classified as hot, only methods calls and returns are registered[17]. The instrumentation of hot methods is made in such a way as to enforce these levels of details so that overhead is limited[17].

Once the information has been gathered, it is forwarded to a cost/benefit model where trace information is analyzed to determine if a particular method is a worthy candidate for parallelization[17]. Of course, the cost/benefit model cannot possibly know what the execution time will be for the method in the future; it only has access to the information that was gathered during the sampling period. As such, the assumption is made that the execution in the future will be equal to the execution time recorded during the sampling period[17]:

$$Sequential\ Execution\ Time: \ \ Exec_s = \#_{Samples} \ast Interval_{Sampling}$$

The estimated execution time for the parallel version is calculated by dividing the sequential execution time by the number of available processors plus the overheads associated with parallelization, such as time spent creating parallel threads and the time spent idle or waiting for dependence to be fulfilled[17]. To estimate the waiting time, the ratio of time spent executing dependent instructions over the execution of the whole method, $$\alpha$$, is used[17]. If this ratio is small enough, the waiting time is approximated to zero[17].

$$Waiting = \begin{cases} 
0 & \text{if } \alpha \leq \frac{1}{\#_{CPU}} \\
\alpha \ast Exec_s - \frac{Exec_s}{\#_{CPU}} & \text{if } \alpha > \frac{1}{\#_{CPU}}
\end{cases}$$

The overhead for the compilation time is estimated as a linear function of the method’s byte-code length[17]. The overhead from the parallel execution environment is simply represented as a constant[17].

$$Overhead = \beta \ast ByteCode_{Length} + \gamma$$

$$Parallel\ Execution\ Time: \ \ Exec_{p} = \frac{Exec_s}{\#_{CPU}} + Waiting + Overhead$$

A method is qualified for parallelization if it satisfies the following inequality[17]:

$$Parallelization\ criteria: \ \ Exec_{p} < Exec_s \ast f$$

The value of $$f$$ can be varied to allow only methods which provide a certain speed-up to be parallelized[17]. For example, to parallelize only methods with a projected speedup of 1.25, $$f$$ should be adjusted to 1/1.25.
The approach was evaluated on a dual-core processor using benchmarks from the Java Grande suite. This is the same suite that was used by Bradel in [14] and in fact, two of the benchmarks are the same. This allows for at least a little direct comparison between the two approaches. For these two benchmarks, the reported speedup is about the same, being even a bit slower (~0.1 speedup difference) for the currently discussed approach. The use of only two cores is a bizarre and limiting decision, especially considering that in the year the paper was published, 2011, 4 to 8 cores systems are quite common. By comparison, [14] dates back to 2007 but already considered up to 4 cores.

3. Targeting GPUs

Approaches discussed so far covered multi-core CPUs. However, in the recent years, graphical processing units (GPUs) have become quite the center of attention for the reason that they possess a massively parallel architecture. Programming these GPUs for general purpose computation was however quite a challenge since any algorithm needed to be mapped to a graphical API, such as openGL to be executed and control flow was limited[18]. Along the years, GPU hardware was extended to offer more customizability and programming models, such as CUDA and OpenCL, were introduced. These factors allowed much easier execution of custom applications on GPUs. However, these advancement where mainly made for C/C++ language, leaving languages such as Java behind. This section presents how GPU power can be harnessed, either manually or automatically, from within Java.

3.1 Background

CUDA is a programming model developed by Nvidia for use by C/C++ programmers to develop software that can run on Nvidia GPUs. In CUDA, the programmer writes two portions of code, one executes on the CPU, like normal programs, and the other executes on the GPU. The latter portion is divided into kernels, which are function that are designed to execute in parallel, in a single instruction multiple thread (SIMT) fashion, on the GPU. Since the memory space of the GPU is not directly accessible by the CPU and vice-versa, the execution of all kernels require the transfer of input data from the main memory to the GPU and of output data from the GPU memory back to the main memory if necessary. Figure 21 shows these general steps. Of course, some kernels might not require input data transfer or might not require output data to be transferred back to the CPU.

The typical way of using CUDA from Java is to go through the Java Native Interface (JNI)[18]. The process involves writing Java code which then calls a C function through JNI which handles memory operations required and then makes the CUDA call to execute the kernel[18]. The execution then comes back to the C function, which receives the data, organizes it for the Java program and finally returns control to the Java program. The process, shown in Figure 22 is long, tedious and prone to error.
3.2 Static Parallelization

JCUDA[18] is a programming interface which has been proposed to simplify the previously mentioned process. While it still requires the programmer to write the CUDA kernel, JCUDA generates all the necessary JNI calls to allocate memory and start data transfers[18]. By doing so, JCUDA not only simplifies the process but also reduces the possibility of errors by abstracting the complexity of transferring data appropriately from the Java space to the GPU from the programmer. While JCUDA does not automatically parallelize the code, it automatizes the allocation and transfer of data for kernels from Java code and makes the idea of running code on a GPU from Java more accessible[18]. For the set of proposed benchmarks, all from the Java Grande suite, JCUDA shows speed-up scaling from around 7 up to 120[18]. The authors note that the speed-up factor attained is largely dependent on the number of synchronization instructions required during execution[18].
The work of Peter Calvert[19][20] further automatized the process by introducing a compiler capable of automatically creating parallel kernels from Java loops. The programmer annotates loops it wants parallelized with a @Parallel tag[19]. The compilation process starts with whatever Java compiler is installed performing the transformation to byte-code and then the parallelizing compiler takes over and perform its operations directly on the produced bytecode[20]. It locates loop identified by the annotations, performs static dependence analysis, and create the equivalent CUDA C code if it is determined that the loop can indeed execute in parallel[19]. Of course, due to characteristics of the Java language that were already mentioned, such as the structure of multi-dimensional arrays, a static approach is limited. The compiler does not perform any optimizations on the produced C++ code and instead directly passes it to the NVidia compiler, which then performs optimization and finally proceeds with the conversion to the assembly language of the GPU[19]. While the approach automatizes both the data transfers and the parallelization of the loops, it still requires input from the programmer and is limited by the lack of run-time information. Looking at speed-up, the approach manages speed-up factors of over 190 when considering kernel execution only and of 18 up to 39 when considering the whole application execution[19]. For the measurement, the author also used two benchmarks, crypt and series, from the Java Grande suite that were also utilized to measure the performance of JCUDA[19]. The results, however, are not comparable as the GPU that was used is not the same.

3.3 Run-time Parallelization

The approach in[21] uses the adaptive optimization system insides the JikesRVM to detect methods which use significant amount of execution time to detect candidates for parallelization. It then takes the data collected on these methods and passes it through a cost/benefit model to determine if execution on the GPU is advantageous or not[21]. If it is, the compiler will generate intermediate representation code for the RapidMind framework and, through JNI calls, trigger the creation of the GPU code[21].

The RapidMind framework with a front-end that takes in C++ code and generates code in an internal intermediate representation and a back-end which takes this IR code, optimizes it and finally compiles it for the desired target platform[21]. RapidMind supports compilation for different GPUs, multi-core CPUs and even the Cell broadband engine[21]. Using RapidMind allows the authors to concentrate on detecting parallelization and dealing with Java-specific issue without having to deal with the problem of creating CUDA kernels. A visual overview of the organization is shown in Figure 23.
The parallelization framework takes as input the hot methods selected by the optimization system inside Jikes and classify loops that are inside those methods [22]. There are four classes of loops: GPU-Implicit, GPU-Explicit, CPU and Multi-pass [22]. A GPU-Implicit loop is a loop that executes on the GPU and for which the control flow is not shown explicitly inside the GPU program [22]. A GPU-Explicit loop is also a loop that will execute on the GPU, but for which the control flow is shown explicitly inside the GPU kernel [22]. The reason for which the control flow cannot be abstracted away inside a GPU-Explicit loop can be, for example, because of a loop-carried dependency. This kind of loop cannot be parallelized for the GPU, but may be part of a GPU-Implicit loop [22]. A CPU loop, like the name implies, is a loop that will execute on the CPU [22]. Multi-pass loops are created to handle cases where multiple data output by a GPU-Implicit loop inside a CPU loop will be used by the GPU-Implicit loop inside the next iteration of the CPU loop [22]. Multi-pass loops avoid the useless data transfer in and out of the GPU memory on all iterations [22]. Several conditions must be checked in order to determine to which category a loop belongs. The restrictions listed below, which will be explained later, are taken, word for word, from [22]:

![Figure 23: Block diagram of the proposed approach, including RapidMind. Note that the approach does not use the RapidMind frontend and instead directly generates the IR code. Taken from [22].](image-url)
**Restriction 1:** The parent of L in the nesting tree must be either CPU or GPU-Implicit.

**Restriction 2:** If the parent of L’ of L is also GPU Implicit, L must be the entire body of L’.

**Restriction 3:** No-Loop carried true data dependences exist between iterations of L.

**Restriction 4:** For each array store $A[i_1, \ldots, i_n]$ inside a GPU-Implicit loop, the dimension of the store $n$ must equal the number of GPU-Implicit loops, and $i_k, k \in \{1, \ldots, n\}$, must be the induction variable of the GPU-Implicit loops, in order of nesting, from the outermost loop to the innermost.

**Restriction 5:** The parent of the loop must be a CPU loop and contain at least one child that is GPU-Implicit.

**Restriction 6:** For each array $A$, array read and writes in the loop must either all be inside GPU-Implicit children of the loops, or all be outside of any GPU-Implicit children of the loop, but not both.

A GPU-Implicit loop must satisfy restriction 1, 2, 3 and 4[22]. The first restriction ensures that the parent of the outermost GPU-Implicit loop is CPU code. This is necessary since the CPU is the one to initiate the change from CPU to GPU. The second and the first restriction together ensure the strict ordering that GPU-Implicit loop are contained inside CPU loop and that GPU-Explicit loops are contained inside GPU-Implicit loops. The third restriction ensures that no dependence exist between iterations. Loop-carried anti-dependences are taken care of by the fact that all array data must be copied to the GPU before the loop starts. Finally, loop-carried output dependences are taken care of by the fourth restriction, which forces each GPU-implicit loop iteration to write only to the memory locations associated with this iteration[22]. A GPU-Explicit loop must be nested inside a GPU-Implicit loop and also abide by restriction 4[22]. Loop carried dependences are allowed inside GPU-Explicit loops[22]. Multi-pass loops must respect restriction 5 and 6[22]. This is to make sure that a multi-pass loop resides outside of all GPU-Implicit loop in the nest, and that any array that is copied to the GPU is used only inside the GPU for the duration of the loop[22]. Figure 24 shows an example of CPU code being transformed into GPU code which contains all categories of loops.

The authors identify four main issues with Java that can become obstacles when parallelizing for GPUs: aliasing, multi-dimensional arrays organization, exceptions, and unstructured control-flow inside bytecode representation.
In Java, all array accesses are done by using the address of the array, by using a pointer to the array. This makes detecting if two variables alias to each other very difficult at compile-time. During dependence analysis, dependences whose existence depends on two references aliasing to each other are marked with the AliasOnly tag[22]. When the parallelization algorithm encounters a method with AliasOnly dependences, it parallelizes it anyways and adds guards to check if, at run-time the relation exists or not[22]. In the case where the dependence exists, a backup CPU version executes instead of the GPU version[22].

Multi-dimensional arrays are a huge problem for any parallelization algorithm in Java. The reason for this is that multi-dimensional arrays are represented as arrays of arrays which make it possible that two arrays alias to each other through a sub-array reference, to have superior dimension arrays that are not equal in length, and to have null references to a sub-array. Not only that, but in byte-code, access to arrays of more than one dimension are represented as multiple instructions, each to a one dimension array[22]. The original index vector thus has to be recovered by analyzing these multiple instructions[22].

Detecting that two multi-dimensional array alias to each other is very time consuming as one has to go through all references in all arrays and compare if any of the two aliases to the same memory region. To get around this problem, the authors suggest adding a “dense” flag to the array definition[22]. This flag is set to true when a rectangular array is initialized, as in there are no null references, otherwise it is set false[22]. If any operation overwrites a sub array, the flag is also set to false[22]. It is impossible in Java to have sub-arrays alias to each other simply by initializing arrays[22]. There must be an explicit write operation to a sub-array for this situation to arise and thus the dense flag will inevitably be set to false for at least one of the two arrays[22]. At run-time, guards evaluate the dense flag and if any array has its dense flag set to false, the loop is executed on the CPU instead[22]. For arrays of more than two dimensions, the dense flag must be checked for all dimensions except the last[22]. While this

![Figure 24](image_url)

Figure 24: CPU code, on the left, transformed for GPU execution, on the right. The types of loops identified in the CPU code are shown. Taken from[22].
process can be long, it is most certainly shorter than checking for all references in all dimensions to find aliases.

All exceptions for null pointers and out of bound access are replaced by guards that are verified at run-time before executing the loops[22]. References are checked to see if they are null and all array index expressions are evaluated to see if a possible access could be made out of the bounds of the array[22]. The parallelization framework supports evaluating constant array index expression as well as linear expression of the $ax + b$ form, where $x$ is the loop induction variable and both $a$ and $b$ are invariants[22]. If an array is accessed through an expression that is not of this form, the framework will simply not parallelize the loops. In the case where an access to a null reference or an out-of-bound access is detected, the CPU version is executed instead of the GPU version[22]. If any explicit throw statement is detected, then the loop is also not parallelized[22].

Finally, a problem of making the analysis at byte-code level is that control flow information is unstructured[22]. This means that all loops and if statements have been replaced by simple goto statements. RapidMind requires structured flow control in its intermediate representation so structured control has to be rebuilt by traversing the control-flow graph[22].

In order to only parallelize loops that offer enough work so that the benefits gained from parallelization can outweigh the cost of parallelization and of all necessary data transfers between main memory and GPU memory, a cost/benefit model is implemented[22]. The model has the following parameters[22]:

- $T_{CPU}, T_{GPU}$: Average time needed per instruction on the CPU/GPU. This value is derived from data collected offline through micro-benchmarks.

- $insts$: Number of instructions that executed in the loop to be parallelized. Several simplifying assumptions are made by the current work: conditional branches all have a 50/50 chance of being taken and all nested loops are assumed to execute for ten iterations except if the total number of iterations is defined by a constant value.

- $A_{out}$: Output array that is to be returned from the GPU memory to the main memory at the end of the kernel. The loop iterates once for all elements of this array.

- $Copy$: Estimated time to copy a single floating point value between the GPU and the CPU memory.

- $A_{inout}$: Set of all input arrays that will need to be copied to GPU memory and of all output arrays that will need to be copied from GPU memory.

- $Init$: Time to set up GPU to execute a kernel. Can vary depending on the GPU used, but is constant for a particular GPU.
From these parameters, an estimated execution cost for both CPU and GPU can be derived[22]:

**Cost on CPU:**

\[ \text{CPU}_{\text{cost}} = T_{CPU} \times \text{insts} \times A_{\text{out.size}} \]

The cost of executing on the CPU corresponds to the average time to execute an instruction times the number of instruction times the number of iteration for the outermost loop.

**Cost on GPU:**

\[ \text{GPU}_{\text{cost}} = T_{GPU} \times \text{insts} \times A_{\text{out.size}} + \text{copy} \times \sum_{A \in A_{\text{inout}}} A_{\text{size}} + \text{init} \]

The cost of executing on the GPU has three parts: time to execute all instructions plus the time to copy all arrays to and from the GPU memory plus the time to initialize the GPU.

If, when comparing these two values, the results show that the execution time on the GPU, accounting for all overheads, is smaller than the cost of executing on the CPU, then the loop will most likely be parallelized[22]. Of course, some factor might also be added, like in[17], to ensure that only those that provide a certain level of speedup, over 4.0 for example, are parallelized.

The speed-up for the set of benchmarks considered, which are not really detailed and were designed by the authors, ranges from 1.27 up to 13 in the ideal case[22]. The ideal case assumes that the performance model was tuned specifically for the benchmark being run[22]. When this was not the case, performance was reduced by 4.5% up to 11.5%[22]. Once again, the GPU used is very different from the two previously mentioned approaches, not only not having the same number of cores, but also being from a different generation of hardware and being compatible with a different version of CUDA. This prevents direction comparison of the results.
4. Overview of Covered Approaches

Table 2: Characteristics of different approaches to parallelizing Java programs on multicore CPUs

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Run-time</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Run-time approach</td>
<td>-</td>
<td>Checks</td>
<td>Pipelined TLS</td>
<td>Speculation</td>
<td>Checks</td>
<td>Speculation</td>
<td>Checks</td>
</tr>
<tr>
<td>Unit of parallelism</td>
<td>Loops</td>
<td>Loops</td>
<td>Loops</td>
<td>Loops</td>
<td>Methods</td>
<td>Traces</td>
<td>Traces</td>
</tr>
<tr>
<td>Hardware requirements</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y1</td>
<td>N</td>
</tr>
<tr>
<td>Cost-Benefit Model</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N2</td>
<td>Y</td>
</tr>
</tbody>
</table>

Table 3: Characteristics of different approaches to parallelizing Java programs on GPUs

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic detection of parallelism</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Automatic data transfers</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Automatic CUDA kernel creation</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Run-time</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Run-time approach</td>
<td>-</td>
<td>-</td>
<td>Checks</td>
<td></td>
</tr>
<tr>
<td>Unit of parallelism</td>
<td>Loops</td>
<td>Loops</td>
<td>Loops</td>
<td></td>
</tr>
<tr>
<td>Hardware support</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Cost-Benefit Model</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Year</td>
<td>2009</td>
<td>2010</td>
<td>2009-2011</td>
<td></td>
</tr>
</tbody>
</table>

1 Yes for the 2010 version which covers both loops and recursive methods and requires transactional memory. No for the original 2007 paper which only covered the parallelization of loops.
2 Using traces in itself applies some kind of cost/benefit selection since traces are frequently executing code. However, the approach does not have an explicit cost/benefits model.
3 Detection of parallelism is automatic within zones identified with annotations by the programmer.
5. Future Work and Conclusion

Automatic parallelization for Java has been achieved on both CPU and GPU with a certain level of success, as was shown by the approaches presented in this report. This section suggests some further work that could be done in this field that was derived from analyzing the aforementioned techniques.

First, the work in [21][22] uses a cost/benefit model to evaluate if running a certain parallel code on the GPU is worthwhile taking into account all the associated overheads. This proves necessary since a piece of code that does not provide enough parallelism might result in slower performance than the sequential code due to these overheads. However, when a piece of code is deemed not to provide enough parallelism, the sequential code is used. Instead, why not use a second, fall-back, cost/benefit model to analyze if instead, it might be worth the trouble to parallelize it for a multi-core CPU? This is an especially simple modification to do in the case of this approach since RapidMind already supports multi-core CPUs. Also, GPUs are good for loop-level parallelism with intensive computations, but they would not be the ideal platform for method-level[13] or trace-level[14][16] parallelism since GPUs use a SIMT approach and their performance degrades rapidly in the presence of diverging control-flow. Adding support for multicore CPUs along with the GPU aspect would enable the JVM to tackle these different types of parallelism.

Secondly, all of the different work presented in this report use rather basic tests for dependence and also use rather basic transformations to ease or eliminate dependences. Extending the amount of transformations supported and implementing more precise tests, such as the Polyhedral and Polytope model[23][24] would be interesting.

Finally, an observation that was mentioned in [21][22], is that, in some case, the inaccuracies in the cost/benefit model resulted in loops getting selected for parallelization whose execution time ended up being worse on the GPU than on the CPU. However, there is no going back, the loop will always from now on execute on the GPU. Adding some adaptivity to the system so that it can revert back decisions that turns out to be bad could lessen the negative impact. Such an adaptive model could be an interesting approach to test.

The approaches covered here have shown that parallelizing Java programs is possible, and viable, at both compile-time and run-time for both CPUs and GPUs, providing speed-up with minimal, if no, intervention for the programmer. The proof of concept has been done and future direction for parallelizing Java on these two hardware platforms will most likely take the form of making the process more efficient, more accurate, more adaptive and more integrated. The number of cores in systems only goes on to increase with no end in sight. As such, the benefits gained from parallelization only go on increasing for the foreseeable future.
References


